

System.Collections.Generic Namespace

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

[+] Expand table

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.
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 >.ValueCollection	Represents the collection of values in a 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.

Structs

[] [Expand table](#)

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>.UnorderedItemsCollection.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

 [Expand table](#)

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.
 IReadOnlyDictionary<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.

Remarks

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<T>](#) is a generic version of [ArrayList](#). There are generic [Queue<T>](#) and [Stack<T>](#) 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<T>](#) generic class is a true linked list and has no nongeneric counterpart.

See also

- [Generic Collections in .NET](#)

CollectionExtensions Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Provides extension methods for generic collections.

C#

```
public static class CollectionExtensions
```

Inheritance [Object](#) → CollectionExtensions

Methods

 Expand table

GetValueOrDefault<TKey,TValue>(IReadOnlyDictionary<TKey,TValue>, TKey, TValue)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
GetValueOrDefault<TKey,TValue>(IReadOnlyDictionary<TKey,TValue>, TKey)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
Remove<TKey,TValue>(IDictionary<TKey,TValue>, TKey, TValue)	Tries to remove the value with the specified <code>key</code> from the <code>dictionary</code> .
TryAdd<TKey,TValue>(IDictionary<TKey,TValue>, TKey, TValue)	Tries to add the specified <code>key</code> and <code>value</code> to the <code>dictionary</code> .

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

CollectionExtensions.GetValueOrDefault Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Overloads

 Expand table

<code>GetValueOrDefault<TKey, TValue></code> <code>(IReadOnlyDictionary<TKey, TValue>, TKey)</code>	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
<code>GetValueOrDefault<TKey, TValue></code> <code>(IReadOnlyDictionary<TKey, TValue>, TKey, TValue)</code>	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .

GetValueOrDefault<TKey, TValue> (IReadOnlyDictionary<TKey, TValue>, TKey)

Tries to get the value associated with the specified `key` in the `dictionary`.

C#

```
public static TValue? GetValueOrDefault<TKey, TValue>(this  
System.Collections.Generic.IReadOnlyDictionary<TKey, TValue> dictionary, TKey  
key);
```

Type Parameters

`TKey`

The type of the keys in the dictionary.

`TValue`

The type of the values in the dictionary.

Parameters

dictionary [IReadOnlyDictionary<TKey,TValue>](#)

A dictionary with keys of type `TKey` and values of type `TValue`.

key `TKey`

The key of the value to get.

Returns

`TValue`

A `TValue` instance. When the method is successful, the returned object is the value associated with the specified `key`. When the method fails, it returns the `defaultValue` value for `TValue`.

Exceptions

[ArgumentNullException](#)

`dictionary` is `null`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

GetValueOrDefault<TKey,TValue> ([IReadOnlyDictionary<TKey,TValue>](#), `TKey`, `TValue`)

Tries to get the value associated with the specified `key` in the `dictionary`.

C#

```
public static TValue GetValueOrDefault<TKey,TValue>(this  
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue> dictionary, TKey  
key, TValue defaultValue);
```

Type Parameters

TKey

The type of the keys in the dictionary.

TValue

The type of the values in the dictionary.

Parameters

dictionary [IReadOnlyDictionary<TKey,TValue>](#)

A dictionary with keys of type `TKey` and values of type `TValue`.

key `TKey`

The key of the value to get.

defaultValue `TValue`

The default value to return when the `dictionary` cannot find a value associated with the specified `key`.

Returns

`TValue`

A `TValue` instance. When the method is successful, the returned object is the value associated with the specified `key`. When the method fails, it returns `defaultValue`.

Exceptions

[ArgumentNullException](#)

`dictionary` is `null`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

CollectionExtensions.

Remove<TKey,TValue> Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Tries to remove the value with the specified `key` from the `dictionary`.

C#

```
public static bool Remove<TKey, TValue>(this  
System.Collections.Generic.IDictionary<TKey, TValue> dictionary, TKey key, out  
TValue value);
```

Type Parameters

TKey

The type of the keys in the `dictionary`.

TValue

The type of the values in the `dictionary`.

Parameters

dictionary [IDictionary<TKey,TValue>](#)

A dictionary with keys of type `TKey` and values of type `TValue`.

key TKey

The key of the value to remove.

value TValue

When this method returns `true`, the removed value; when this method returns `false`, the `default` value for `TValue`.

Returns

[Boolean](#)

`true` when a value is found in the `dictionary` with the specified `key`; `false` when the `dictionary` cannot find a value associated with the specified `key`.

Exceptions

ArgumentNullException

`dictionary` is `null`.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

CollectionExtensions.TryAdd<TKey,TValue> Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Tries to add the specified `key` and `value` to the `dictionary`.

C#

```
public static bool TryAdd<TKey, TValue>(this  
System.Collections.Generic.IDictionary<TKey, TValue> dictionary, TKey key, TValue  
value);
```

Type Parameters

TKey

The type of the keys in the dictionary.

TValue

The type of the values in the dictionary.

Parameters

dictionary [IDictionary<TKey,TValue>](#)

A dictionary with keys of type `TKey` and values of type `TValue`.

key `TKey`

The key of the value to add.

value `TValue`

The value to add.

Returns

Boolean

`true` when the `key` and `value` are successfully added to the `dictionary`; `false` when the `dictionary` already contains the specified `key`, in which case nothing gets added.

Exceptions

ArgumentNullException

`dictionary` is `null`.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Comparer<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Provides a base class for implementations of the [IComparer<T>](#) generic interface.

C#

```
public abstract class Comparer<T> : System.Collections.Generic.IComparer<T>,
System.Collections.IComparer
```

Type Parameters

T

The type of objects to compare.

Inheritance [Object](#) → Comparer<T>

Implements [IComparer<T>](#) , [IComparer](#)

Examples

The following example derives a class, `BoxLengthFirst`, from the [Comparer<T>](#) class. This comparer compares two objects of type `Box`. It sorts them first by length, then by height, and then by width. The `Box` class implements the [IComparable<T>](#) interface to control the default comparison between two `Box` objects. This default implementation sorts first by height, then by length, and then by width. The example shows the differences between the two comparisons by sorting a list of `Box` objects first by using the `BoxLengthFirst` comparer and then by using the default comparer.

C#

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

class Program
{
    static void Main(string[] args)
```

```
{  
    List<Box> Boxes = new List<Box>();  
    Boxes.Add(new Box(4, 20, 14));  
    Boxes.Add(new Box(12, 12, 12));  
    Boxes.Add(new Box(8, 20, 10));  
    Boxes.Add(new Box(6, 10, 2));  
    Boxes.Add(new Box(2, 8, 4));  
    Boxes.Add(new Box(2, 6, 8));  
    Boxes.Add(new Box(4, 12, 20));  
    Boxes.Add(new Box(18, 10, 4));  
    Boxes.Add(new Box(24, 4, 18));  
    Boxes.Add(new Box(10, 4, 16));  
    Boxes.Add(new Box(10, 2, 10));  
    Boxes.Add(new Box(6, 18, 2));  
    Boxes.Add(new Box(8, 12, 4));  
    Boxes.Add(new Box(12, 10, 8));  
    Boxes.Add(new Box(14, 6, 6));  
    Boxes.Add(new Box(16, 6, 16));  
    Boxes.Add(new Box(2, 8, 12));  
    Boxes.Add(new Box(4, 24, 8));  
    Boxes.Add(new Box(8, 6, 20));  
    Boxes.Add(new Box(18, 18, 12));  
  
    // Sort by an Comparer<T> implementation that sorts  
    // first by the length.  
    Boxes.Sort(new BoxLengthFirst());  
  
    Console.WriteLine("H - L - W");  
    Console.WriteLine("=====");  
    foreach (Box bx in Boxes)  
    {  
        Console.WriteLine("{0}\t{1}\t{2}",  
            bx.Height.ToString(), bx.Length.ToString(),  
            bx.Width.ToString());  
    }  
  
    Console.WriteLine();  
    Console.WriteLine("H - L - W");  
    Console.WriteLine("=====");  
  
    // Get the default comparer that  
    // sorts first by the height.  
    Comparer<Box> defComp = Comparer<Box>.Default;  
  
    // Calling Boxes.Sort() with no parameter  
    // is the same as calling Boxes.Sort(defComp)  
    // because they are both using the default comparer.  
    Boxes.Sort();  
  
    foreach (Box bx in Boxes)  
    {  
        Console.WriteLine("{0}\t{1}\t{2}",  
            bx.Height.ToString(), bx.Length.ToString(),  
            bx.Width.ToString());  
    }  
}
```

```
// This explicit interface implementation
// compares first by the length.
// Returns -1 because the length of BoxA
// is less than the length of BoxB.
BoxLengthFirst LengthFirst = new BoxLengthFirst();

Comparer<Box> bc = (Comparer<Box>) LengthFirst;

Box BoxA = new Box(2, 6, 8);
Box BoxB = new Box(10, 12, 14);
int x = LengthFirst.Compare(BoxA, BoxB);
Console.WriteLine();
Console.WriteLine(x.ToString());
}

}

public class BoxLengthFirst : Comparer<Box>
{
    // Compares by Length, Height, and Width.
    public override int Compare(Box x, Box y)
    {
        if (x.Length.CompareTo(y.Length) != 0)
        {
            return x.Length.CompareTo(y.Length);
        }
        else if (x.Height.CompareTo(y.Height) != 0)
        {
            return x.Height.CompareTo(y.Height);
        }
        else if (x.Width.CompareTo(y.Width) != 0)
        {
            return x.Width.CompareTo(y.Width);
        }
        else
        {
            return 0;
        }
    }
}

// This class is not demonstrated in the Main method
// and is provided only to show how to implement
// the interface. It is recommended to derive
// from Comparer<T> instead of implementing IComparer<T>.
public class BoxComp : IComparer<Box>
{
    // Compares by Height, Length, and Width.
    public int Compare(Box x, Box y)
    {
        if (x.Height.CompareTo(y.Height) != 0)
        {
            return x.Height.CompareTo(y.Height);
        }
    }
}
```

```

        else if (x.Length.CompareTo(y.Length) != 0)
    {
        return x.Length.CompareTo(y.Length);
    }
    else if (x.Width.CompareTo(y.Width) != 0)
    {
        return x.Width.CompareTo(y.Width);
    }
    else
    {
        return 0;
    }
}

public class Box : IComparable<Box>
{
    public Box(int h, int l, int w)
    {
        this.Height = h;
        this.Length = l;
        this.Width = w;
    }
    public int Height { get; private set; }
    public int Length { get; private set; }
    public int Width { get; private set; }

    public int CompareTo(Box other)
    {
        // Compares Height, Length, and Width.
        if (this.Height.CompareTo(other.Height) != 0)
        {
            return this.Height.CompareTo(other.Height);
        }
        else if (this.Length.CompareTo(other.Length) != 0)
        {
            return this.Length.CompareTo(other.Length);
        }
        else if (this.Width.CompareTo(other.Width) != 0)
        {
            return this.Width.CompareTo(other.Width);
        }
        else
        {
            return 0;
        }
    }
}

```

Remarks

Derive from this class to provide a custom implementation of the `IComparer<T>` interface for use with collection classes such as the `SortedList< TKey, TValue >` and `SortedDictionary< TKey, TValue >` generic classes.

The difference between deriving from the `Comparer<T>` class and implementing the `System.IComparable` interface is as follows:

- To specify how two objects should be compared by default, implement the `System.IComparable` interface in your class. This ensures that sort operations will use the default comparison code that you provided.
- To define a comparer to use instead of the default comparer, derive from the `Comparer<T>` class. You can then use this comparer in sort operations that take a comparer as a parameter.

The object returned by the `Default` property uses the `System.IComparable<T>` generic interface (`IComparable<T>` in C#, `IComparable(Of T)` in Visual Basic) to compare two objects. If type `T` does not implement the `System.IComparable<T>` generic interface, the `Default` property returns a `Comparer<T>` that uses the `System.IComparable` interface.

Notes to Implementers

`Compare(T, T)` and `Equals(T, T)` may behave differently in terms of culture-sensitivity and case-sensitivity.

For string comparisons, the `StringComparer` class is recommended over `Comparer<String>`. Properties of the `StringComparer` class return predefined instances that perform string comparisons with different combinations of culture-sensitivity and case-sensitivity. The case-sensitivity and culture-sensitivity are consistent among the members of the same `StringComparer` instance.

For more information on culture-specific comparisons, see the `System.Globalization` namespace and [Globalization and Localization](#).

Constructors

 [Expand table](#)

<code>Comparer<T>()</code>	Initializes a new instance of the <code>Comparer<T></code> class.
----------------------------------	---

Properties

[Expand table](#)

Default	Returns a default sort-order comparer for the type specified by the generic argument.
---------	---

Methods

[Expand table](#)

Compare(T, T)	When overridden in a derived class, performs a comparison of two objects of the same type and returns a value indicating whether one object is less than, equal to, or greater than the other.
Create(Comparison<T>)	Creates a comparer by using the specified comparison.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
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)
ToString()	Returns a string that represents the current object. (Inherited from Object)

Explicit Interface Implementations

[Expand table](#)

IComparer.Compare(Object, Object)	Compares two objects and returns a value indicating whether one is less than, equal to, or greater than the other.
---	--

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IComparer<T>](#)
- [IComparable<T>](#)
- [StringComparer](#)
- [CultureInfo](#)

Comparer<T> Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [Comparer<T>](#) class.

C#

```
protected Comparer();
```

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Comparer<T>.Default Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns a default sort-order comparer for the type specified by the generic argument.

C#

```
public static System.Collections.Generic.Comparer<T> Default { get; }
```

Property Value

[Comparer<T>](#)

An object that inherits [Comparer<T>](#) and serves as a sort-order comparer for type [T](#).

Examples

The following example shows how to use the [Default](#) property to get an object that performs the default comparison. This example is part of a larger example provided for the [Comparer<T>](#) class.

C#

```
// Get the default comparer that
// sorts first by the height.
Comparer<Box> defComp = Comparer<Box>.Default;

// Calling Boxes.Sort() with no parameter
// is the same as calling Boxes.Sort(defComp)
// because they are both using the default comparer.
Boxes.Sort();

foreach (Box bx in Boxes)
{
    Console.WriteLine("{0}\t{1}\t{2}",
        bx.Height.ToString(), bx.Length.ToString(),
        bx.Width.ToString());
}
```

Remarks

The [Comparer<T>](#) returned by this property uses the [System.IComparable<T>](#) generic interface ([IComparable<T>](#) in C#, [IComparable\(Of T\)](#) in Visual Basic) to compare two objects. If type `T` does not implement the [System.IComparable<T>](#) generic interface, this property returns a [Comparer<T>](#) that uses the [System.IComparable](#) interface.

Notes to Callers

For string comparisons, the [StringComparer](#) class is recommended over [Comparer<String>](#) ([Comparer\(Of String\)](#) in Visual Basic). Properties of the [StringComparer](#) class return predefined instances that perform string comparisons with different combinations of culture-sensitivity and case-sensitivity. The case-sensitivity and culture-sensitivity are consistent among the members of the same [StringComparer](#) instance.

For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IComparable<T>](#)
- [IComparable](#)
- [StringComparer](#)
- [Object](#)

Comparer<T>.Compare(T, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

When overridden in a derived class, performs a comparison of two objects of the same type and returns a value indicating whether one object is less than, equal to, or greater than the other.

C#

```
public abstract int Compare(T? x, T? y);
```

Parameters

x **T**

The first object to compare.

y **T**

The second object to compare.

Returns

[Int32](#)

A signed integer that indicates the relative values of **x** and **y**, as shown in the following table.

 [Expand table](#)

Value	Meaning
Less than zero	x is less than y .
Zero	x equals y .
Greater than zero	x is greater than y .

Implements

[Compare\(T, T\)](#)

Exceptions

ArgumentException

Type `T` does not implement either the `IComparable<T>` generic interface or the `IComparable` interface.

Examples

The following example defines a comparer of `Box` objects that can be used instead of the default comparer. This example is part of a larger example provided for the `Comparer<T>` class.

C#

```
public class BoxLengthFirst : Comparer<Box>
{
    // Compares by Length, Height, and Width.
    public override int Compare(Box x, Box y)
    {
        if (x.Length.CompareTo(y.Length) != 0)
        {
            return x.Length.CompareTo(y.Length);
        }
        else if (x.Height.CompareTo(y.Height) != 0)
        {
            return x.Height.CompareTo(y.Height);
        }
        else if (x.Width.CompareTo(y.Width) != 0)
        {
            return x.Width.CompareTo(y.Width);
        }
        else
        {
            return 0;
        }
    }
}
```

Remarks

Implement this method to provide a customized sort order comparison for type `T`.

Notes to Implementers

Comparing `null` with any reference type is allowed and does not generate an exception. A null reference is considered to be less than any reference that is not null.

For information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IComparable<T>](#)
- [IComparable](#)
- [StringComparer](#)
- [Object](#)
- [CultureInfo](#)

Comparer<T>.Create(Comparison<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Creates a comparer by using the specified comparison.

C#

```
public static System.Collections.Generic.Comparer<T> Create(Comparison<T>
comparison);
```

Parameters

comparison [Comparison<T>](#)

The comparison to use.

Returns

[Comparer<T>](#)

The new comparer.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Comparer<T>.IComparer.Compare(Object, Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Compares two objects and returns a value indicating whether one is less than, equal to, or greater than the other.

C#

```
int IComparer.Compare(object x, object y);
```

Parameters

x [Object](#)

The first object to compare.

y [Object](#)

The second object to compare.

Returns

[Int32](#)

A signed integer that indicates the relative values of **x** and **y**, as shown in the following table.

[] [Expand table](#)

Value	Meaning
Less than zero	x is less than y .
Zero	x equals y .
Greater than zero	x is greater than y .

Implements

[Compare\(Object, Object\)](#)

Exceptions

ArgumentException

`x` or `y` is of a type that cannot be cast to type `T`.

-or-

`x` and `y` do not implement either the `IComparable<T>` generic interface or the `IComparable` interface.

Examples

The following example shows how to use the `IComparer.Compare` method to compare two objects. This example is part of a larger example provided for the `Comparer<T>` class.

C#

```
// This explicit interface implementation
// compares first by the length.
// Returns -1 because the length of BoxA
// is less than the length of BoxB.
BoxLengthFirst LengthFirst = new BoxLengthFirst();

Comparer<Box> bc = (Comparer<Box>) LengthFirst;

Box BoxA = new Box(2, 6, 8);
Box BoxB = new Box(10, 12, 14);
int x = LengthFirst.Compare(BoxA, BoxB);
Console.WriteLine();
Console.WriteLine(x.ToString());
```

Remarks

This method is a wrapper for the `Compare(T, T)` method, so `obj` must be cast to the type specified by the generic argument `T` of the current instance. If it cannot be cast to `T`, an `ArgumentException` is thrown.

Comparing `null` with any reference type is allowed and does not generate an exception. When sorting, `null` is considered to be less than any other object.

Notes to Callers

`Compare(T, T)` and `Equals(T, T)` behave differently in terms of culture-sensitivity and case-sensitivity.

For string comparisons, the [StringComparer](#) class is recommended over `Comparer<String>`.

Properties of the [StringComparer](#) class return predefined instances that perform string comparisons with different combinations of culture-sensitivity and case-sensitivity. The case-sensitivity and culture-sensitivity are consistent among the members of the same [StringComparer](#) instance.

For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [StringComparer](#)
- [IComparer](#)
- [IComparable](#)
- [CurrentCulture](#)
- [CultureInfo](#)

Dictionary<TKey,TValue>.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [Dictionary<TKey,TValue>](#).

C#

```
public struct Dictionary<TKey,TValue>.Enumerator :  
    System.Collections.Generic.IEnumerator<System.Collections.Generic.KeyValuePair<TKey,TValue>>, System.Collections.IDictionaryEnumerator
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [ValueType](#) → [Dictionary<TKey,TValue>.Enumerator](#)

Implements [IEnumerator<KeyValuePair<TKey,TValue>>](#), [IDictionaryEnumerator](#),
[IEnumerator](#), [IDisposable](#)

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

`Current` returns the same object until `MoveNext` is called. `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this position, subsequent calls to `MoveNext` also return `false`. If the last call to `MoveNext` returned `false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to `MoveNext` or `IEnumerator.Reset` throws an `InvalidOperationException`.

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are `Remove` and `Clear`.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in `System.Collections.Generic` are not synchronized.

Properties

[+] Expand table

<code>Current</code>	Gets the element at the current position of the enumerator.
----------------------	---

Methods

[+] Expand table

<code>Dispose()</code>	Releases all resources used by the <code>Dictionary<TKey,TValue>.Enumerator</code> .
------------------------	--

<code>MoveNext()</code>	Advances the enumerator to the next element of the <code>Dictionary<TKey,TValue></code> .
-------------------------	---

Explicit Interface Implementations

IDictionaryEnumerator.Entry	Gets the element at the current position of the enumerator.
IDictionaryEnumerator.Key	Gets the key of the element at the current position of the enumerator.
IDictionaryEnumerator.Value	Gets the value of the element at the current position of the enumerator.
IEnumerator.Current	Gets the element at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [Dictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

Dictionary< TKey, TValue >.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public System.Collections.Generic.KeyValuePair< TKey, TValue > Current { get; }
```

Property Value

[KeyValuePair< TKey, TValue >](#)

The element in the [Dictionary< TKey, TValue >](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)

Dictionary< TKey, TValue >.Enumerator. Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [Dictionary< TKey, TValue >.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [Dictionary<TKey,TValue>](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to [MoveNext](#) advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

Dictionary<TKey,TValue>.Enumerator.IDictionaryEnumerator.Entry Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
System.Collections.DictionaryEntry System.Collections.IDictionaryEnumerator.Entry  
{ get; }
```

Property Value

[DictionaryEntry](#)

The element in the dictionary at the current position of the enumerator, as a [DictionaryEntry](#).

Implements

[Entry](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IDictionaryEnumerator.Entry](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IDictionaryEnumerator.Entry](#) property.

- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IDictionaryEnumerator.Entry](#) does not move the position of the enumerator, and consecutive calls to [IDictionaryEnumerator.Entry](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

Dictionary<TKey,TValue>.Enumerator.IDictionaryEnumerator.Key Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the key of the element at the current position of the enumerator.

C#

```
object System.Collections.IDictionaryEnumerator.Key { get; }
```

Property Value

[Object](#)

The key of the element in the dictionary at the current position of the enumerator.

Implements

[Key](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IDictionaryEnumerator.Key](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IDictionaryEnumerator.Key](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IDictionaryEnumerator.Key](#) does not move the position of the enumerator, and consecutive calls to [IDictionaryEnumerator.Key](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

Dictionary<TKey, TValue>.Enumerator.IDictionaryEnumerator.Value Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the value of the element at the current position of the enumerator.

C#

```
object? System.Collections.IDictionaryEnumerator.Value { get; }
```

Property Value

[Object](#)

The value of the element in the dictionary at the current position of the enumerator.

Implements

[Value](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IDictionaryEnumerator.Value](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IDictionaryEnumerator.Value](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IDictionaryEnumerator.Value](#) does not move the position of the enumerator, and consecutive calls to [IDictionaryEnumerator.Value](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

Dictionary< TKey, TValue >.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator, as an [Object](#).

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

`IEnumerator.Current` does not move the position of the enumerator, and consecutive calls to `IEnumerator.Current` return the same object until either `MoveNext` or `IEnumerator.Reset` is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

Dictionary<TKey, TValue>.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling the [IEnumerator.Reset](#) method, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

Dictionary<TKey,TValue>.KeyCollection.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [Dictionary<TKey,TValue>.KeyCollection](#).

C#

```
public struct Dictionary<TKey,TValue>.KeyCollection.Enumerator :  
System.Collections.Generic.IEnumerator<TKey>
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [ValueType](#) → [Dictionary<TKey,TValue>.KeyCollection.Enumerator](#)

Implements [IEnumerator<TKey>](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Properties

 Expand table

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

 Expand table

Dispose()	Releases all resources used by the Dictionary<TKey,TValue>.KeyCollection.Enumerator .
MoveNext()	Advances the enumerator to the next element of the Dictionary<TKey,TValue>.KeyCollection .

Explicit Interface Implementations

 Expand table

IEnumerator.Current	Gets the element at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.Enumerator](#)
- [Dictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

Dictionary<TKey,TValue>.KeyCollection.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public TKey Current { get; }
```

Property Value

TKey

The element in the [Dictionary<TKey,TValue>.KeyCollection](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

Dictionary< TKey, TValue >.KeyCollection.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [Dictionary< TKey, TValue >.KeyCollection.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Remarks

To be added

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dispose\(\)](#)
- [Finalize\(\)](#)
- [Cleaning Up Unmanaged Resources](#)

Dictionary<TKey,TValue>.KeyCollection.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [Dictionary<TKey,TValue>.KeyCollection](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to [MoveNext](#) advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

Dictionary<TKey, TValue>.KeyCollection.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

`IEnumerator.Current` does not move the position of the enumerator, and consecutive calls to `IEnumerator.Current` return the same object until either `MoveNext` or `IEnumerator.Reset` is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

Dictionary< TKey, TValue >.KeyCollection.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling the [IEnumerator.Reset](#) method, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

Dictionary<TKey,TValue>.KeyCollection Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents the collection of keys in a [Dictionary<TKey,TValue>](#). This class cannot be inherited.

C#

```
public sealed class Dictionary<TKey, TValue>.KeyCollection :  
    System.Collections.Generic.ICollection<TKey>,  
    System.Collections.Generic.IEnumerable<TKey>,  
    System.Collections.Generic.IReadOnlyCollection<TKey>,  
    System.Collections.ICollection
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → Dictionary<TKey,TValue>.KeyCollection

Implements [ICollection<TKey>](#) , [IEnumerable<T>](#) , [IEnumerable<TKey>](#) ,
[IReadOnlyCollection<TKey>](#) , [ICollection](#) , [IEnumerable](#)

Remarks

The [Dictionary<TKey,TValue>.Keys](#) property returns an instance of this type, containing all the keys in that [Dictionary<TKey,TValue>](#). The order of the keys in the [Dictionary<TKey,TValue>.KeyCollection](#) is unspecified, but it is the same order as the associated values in the [Dictionary<TKey,TValue>.ValueCollection](#) returned by the [Dictionary<TKey,TValue>.Values](#) property.

The [Dictionary<TKey,TValue>.KeyCollection](#) is not a static copy; instead, the [Dictionary<TKey,TValue>.KeyCollection](#) refers back to the keys in the original

[Dictionary<TKey,TValue>](#). Therefore, changes to the [Dictionary<TKey,TValue>](#) continue to be reflected in the [Dictionary<TKey,TValue>.KeyCollection](#).

Constructors

[+] Expand table

Dictionary<TKey,TValue>.KeyCollection(Dictionary<TKey,TValue>)	Initializes a new instance of the Dictionary<TKey,TValue>.KeyCollection class that reflects the keys in the specified Dictionary<TKey,TValue> .
--	---

Properties

[+] Expand table

Count	Gets the number of elements contained in the Dictionary<TKey,TValue>.KeyCollection .
-----------------------	--

Methods

[+] Expand table

CopyTo(TKey[], Int32)	Copies the Dictionary<TKey,TValue>.KeyCollection elements 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 enumerator that iterates through the Dictionary<TKey,TValue>.KeyCollection .
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)
ToString()	Returns a string that represents the current object. (Inherited from Object)

Explicit Interface Implementations

[+] Expand table

<code>ICollection.CopyTo(Array, Int32)</code>	Copies the elements of the <code>ICollection</code> to an <code>Array</code> , starting at a particular <code>Array</code> index.
<code>ICollection.IsSynchronized</code>	Gets a value indicating whether access to the <code>ICollection</code> is synchronized (thread safe).
<code>ICollection.SyncRoot</code>	Gets an object that can be used to synchronize access to the <code>ICollection</code> .
<code>ICollection<TKey>.Add(TKey)</code>	Adds an item to the <code>ICollection<T></code> . This implementation always throws <code>NotSupportedException</code> .
<code>ICollection<TKey>.Clear()</code>	Removes all items from the <code>ICollection<T></code> . This implementation always throws <code>NotSupportedException</code> .
<code>ICollection<TKey>.Contains(TKey)</code>	Determines whether the <code>ICollection<T></code> contains a specific value.
<code>ICollection<TKey>.IsReadOnly</code>	Gets a value indicating whether the <code>ICollection<T></code> is read-only.
<code>ICollection<TKey>.Remove(TKey)</code>	Removes the first occurrence of a specific object from the <code>ICollection<T></code> . This implementation always throws <code>NotSupportedException</code> .
<code>IEnumerable.GetEnumerator()</code>	Returns an enumerator that iterates through a collection.
<code>IEnumerable<TKey>.Get Enumerator()</code>	Returns an enumerator that iterates through a collection.

Extension Methods

[+] Expand table

<code>ToImmutableArray<TSource>(IEnumerable<TSource>)</code>	Creates an immutable array from the specified collection.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.

<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its

	contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)	Applies an accumulator function over a sequence.
Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether all elements of a sequence satisfy a condition.
Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether any element of a sequence satisfies a condition.
Any<TSource>(IEnumerable<TSource>)	Determines whether a sequence contains any elements.
Append<TSource>(IEnumerable<TSource>, TSource)	Appends a value to the end of the sequence.
AsEnumerable<TSource>(IEnumerable<TSource>)	Returns the input typed as

	<code>IEnumerable<T>.</code>
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified

	<code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified

	<p>condition.</p>
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a

	<p>result value from each group and its key. The keys are compared by using a specified comparer.</p>
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	<p>Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	<p>Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	<p>Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	<p>Produces the set intersection of two sequences by using the default equality comparer to compare values.</p>

<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a

	sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.

<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Decimal</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Double</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Int32</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Int64</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable <code>Decimal</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable <code>Double</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.

<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEqualityComparer<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing

	their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then

	returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is

	true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.

ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.

InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`Shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [Dictionary<TKey,TValue>.KeyCollection](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Dictionary<TKey,TValue>.KeyCollection Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [Dictionary<TKey,TValue>.KeyCollection](#) class that reflects the keys in the specified [Dictionary<TKey,TValue>](#).

C#

```
public KeyCollection(System.Collections.Generic.Dictionary<TKey, TValue>
dictionary);
```

Parameters

dictionary [Dictionary<TKey,TValue>](#)

The [Dictionary<TKey,TValue>](#) whose keys are reflected in the new [Dictionary<TKey,TValue>.KeyCollection](#).

Exceptions

[ArgumentNullException](#)

dictionary is `null`.

Remarks

The [Dictionary<TKey,TValue>.KeyCollection](#) is not a static copy; instead, the [Dictionary<TKey,TValue>.KeyCollection](#) refers back to the keys in the original [Dictionary<TKey,TValue>](#). Therefore, changes to the [Dictionary<TKey,TValue>](#) continue to be reflected in the [Dictionary<TKey,TValue>.KeyCollection](#).

This constructor is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.KeyCollection.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements contained in the [Dictionary<TKey,TValue>.KeyCollection](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [Dictionary<TKey,TValue>.KeyCollection](#).

Retrieving the value of this property is an O(1) operation.

Implements

[Count](#) , [Count](#) , [Count](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.KeyCollection.CopyTo(TKey[], Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [Dictionary<TKey,TValue>.KeyCollection](#) elements to an existing one-dimensional [Array](#), starting at the specified array index.

C#

```
public void CopyTo(TKey[] array, int index);
```

Parameters

array TKey[]

The one-dimensional [Array](#) that is the destination of the elements copied from [Dictionary<TKey,TValue>.KeyCollection](#). The [Array](#) must have zero-based indexing.

index Int32

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

The number of elements in the source [Dictionary<TKey,TValue>.KeyCollection](#) is greater than the available space from [index](#) to the end of the destination [array](#).

Remarks

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [Dictionary<TKey,TValue>.KeyCollection](#).

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Array](#)

Dictionary< TKey, TValue >.KeyCollection.Get Enumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [Dictionary< TKey, TValue >.KeyCollection](#).

C#

```
public System.Collections.Generic.Dictionary< TKey, TValue >.KeyCollection.Enumerator  
GetEnumerator();
```

Returns

[Dictionary< TKey, TValue >.KeyCollection.Enumerator](#)

A [Dictionary< TKey, TValue >.KeyCollection.Enumerator](#) for the [Dictionary< TKey, TValue >.KeyCollection](#).

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, `Current` is undefined. You must call `MoveNext` to advance the enumerator to the first element of the collection before reading the value of `Current`.

`Current` returns the same object until `MoveNext` is called. `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this position, subsequent calls to `MoveNext` also return `false`. If the last call to `MoveNext` returned

`false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to `MoveNext` or `IEnumerator.Reset` throws an `InvalidOperationException`.

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are `Remove` and `Clear`.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in `System.Collections.Generic` are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [IEnumerator<T>](#)

Dictionary<TKey,TValue>.KeyCollection. ICollection<TKey>.Add Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to the [ICollection<T>](#). This implementation always throws [NotSupportedException](#).

C#

```
void ICollection<TKey>.Add(TKey item);
```

Parameters

item TKey

The object to add to the [ICollection<T>](#).

Implements

[Add\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

Dictionary<TKey, TValue>.KeyCollection. ICollection<TKey>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all items from the [ICollection<T>](#). This implementation always throws [NotSupportedException](#).

C#

```
void ICollection<TKey>.Clear();
```

Implements

[Clear\(\)](#)

Exceptions

[NotSupportedException](#)

Always thrown.

Remarks

[Count](#) is set to zero, and references to other objects from elements of the collection are also released.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

Dictionary<TKey,TValue>.KeyCollection. ICollection<TKey>.Contains Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [ICollection<T>](#) contains a specific value.

C#

```
bool ICollection<TKey>.Contains(TKey item);
```

Parameters

item TKey

The object to locate in the [ICollection<T>](#).

Returns

Boolean

`true` if `item` is found in the [ICollection<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas, [Dictionary<TKey,TValue>](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.KeyCollection. ICollection<TKey>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool System.Collections.Generic.ICollection<TKey>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [Dictionary<TKey,TValue>.KeyCollection](#), this property always returns `true`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

Dictionary<TKey,TValue>.KeyCollection. ICollection<TKey>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a specific object from the [ICollection<T>](#). This implementation always throws [NotSupportedException](#).

C#

```
bool ICollection<TKey>.Remove(TKey item);
```

Parameters

item TKey

The object to remove from the [ICollection<T>](#).

Returns

Boolean

`true` if `item` was successfully removed from the [ICollection<T>](#); otherwise, `false`. This method also returns `false` if item was not found in the original [ICollection<T>](#).

Implements

[Remove\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown.

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas, [Dictionary<TKey,TValue>](#) allows the user to specify the [IComparer<T>](#)

implementation to use for comparing keys.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

Dictionary<TKey,TValue>.KeyCollection. IEnumerable< TKey >.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerator< TKey > IEnumerable< TKey >.GetEnumerator();
```

Returns

[IEnumerator< TKey >](#)

An [IEnumerator< T >](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

Dictionary< TKey, TValue >.KeyCollection. ICollection.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the non-generic implementations of [ICollection.CopyTo](#) throw [InvalidOperationException](#), whereas the generic implementations throw [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)
- [Array](#)

Dictionary< TKey, TValue >.KeyCollection. ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [Dictionary< TKey, TValue >.KeyCollection](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#).

Synchronization is effective only if all threads lock this object before accessing the collection.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

Dictionary< TKey, TValue >.KeyCollection. ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [Dictionary< TKey, TValue >.KeyCollection](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock this object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;  
lock (ic.SyncRoot) {  
    // Access the collection.  
}
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

Dictionary< TKey, TValue >.KeyCollection. IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator](#)

Dictionary<TKey,TValue>.ValueCollection.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [Dictionary<TKey,TValue>.ValueCollection](#).

C#

```
public struct Dictionary<TKey,TValue>.ValueCollection.Enumerator :  
System.Collections.Generic.IEnumerator<TValue>
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [ValueType](#) → [Dictionary<TKey,TValue>.ValueCollection.Enumerator](#)

Implements [IEnumerator<TValue>](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Properties

[+] Expand table

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

[+] Expand table

Dispose()	Releases all resources used by the Dictionary<TKey,TValue>.ValueCollection.Enumerator .
MoveNext()	Advances the enumerator to the next element of the Dictionary<TKey,TValue>.ValueCollection .

Explicit Interface Implementations

[+] Expand table

IEnumerator.Current	Gets the element at the current position of the enumerator.
-------------------------------------	---

IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.
-------------------------------------	---

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.Enumerator](#)
- [Dictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

Dictionary< TKey, TValue >.ValueCollection.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public TValue Current { get; }
```

Property Value

TValue

The element in the [Dictionary< TKey, TValue >.ValueCollection](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

Dictionary< TKey, TValue >.ValueCollection.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [Dictionary< TKey, TValue >.ValueCollection.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.ValueCollection.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [Dictionary<TKey,TValue>.ValueCollection](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to [MoveNext](#) advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

Dictionary< TKey, TValue >.ValueCollection.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

`IEnumerator.Current` does not move the position of the enumerator, and consecutive calls to `IEnumerator.Current` return the same object until either `MoveNext` or `IEnumerator.Reset` is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

Dictionary< TKey, TValue >.ValueCollection.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling the [IEnumerator.Reset](#) method, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

Dictionary<TKey,TValue>.ValueCollection Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents the collection of values in a [Dictionary<TKey,TValue>](#). This class cannot be inherited.

C#

```
public sealed class Dictionary<TKey, TValue>.ValueCollection :  
    System.Collections.Generic.ICollection<TValue>,  
    System.Collections.Generic.IEnumerable<TValue>,  
    System.Collections.Generic.IReadOnlyCollection<TValue>,  
    System.Collections.ICollection
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → Dictionary<TKey,TValue>.ValueCollection

Implements [ICollection<TValue>](#) , [IEnumerable<T>](#) , [IEnumerable<TValue>](#) ,
[IReadOnlyCollection<TValue>](#) , [ICollection](#) , [IEnumerable](#)

Remarks

The [Dictionary<TKey,TValue>.Values](#) property returns an instance of this type, containing all the values in that [Dictionary<TKey,TValue>](#). The order of the values in the [Dictionary<TKey,TValue>.ValueCollection](#) is unspecified, but it is the same order as the associated keys in the [Dictionary<TKey,TValue>.KeyCollection](#) returned by the [Dictionary<TKey,TValue>.Keys](#) property.

The [Dictionary<TKey,TValue>.ValueCollection](#) is not a static copy; instead, the [Dictionary<TKey,TValue>.ValueCollection](#) refers back to the values in the original

`Dictionary< TKey, TValue >`. Therefore, changes to the `Dictionary< TKey, TValue >` continue to be reflected in the `Dictionary< TKey, TValue >.ValueCollection`.

Constructors

[+] Expand table

<code>Dictionary< TKey, TValue >.ValueCollection(Dictionary< TKey, TValue >)</code>	Initializes a new instance of the <code>Dictionary< TKey, TValue >.ValueCollection</code> class that reflects the values in the specified <code>Dictionary< TKey, TValue ></code> .
---	---

Properties

[+] Expand table

<code>Count</code>	Gets the number of elements contained in the <code>Dictionary< TKey, TValue >.ValueCollection</code> .
--------------------	--

Methods

[+] Expand table

<code>CopyTo(TValue[], Int32)</code>	Copies the <code>Dictionary< TKey, TValue >.ValueCollection</code> elements to an existing one-dimensional <code>Array</code> , starting at the specified array index.
<code>Equals(Object)</code>	Determines whether the specified object is equal to the current object. (Inherited from <code>Object</code>)
<code>GetEnumerator()</code>	Returns an enumerator that iterates through the <code>Dictionary< TKey, TValue >.ValueCollection</code> .
<code>GetHashCode()</code>	Serves as the default hash function. (Inherited from <code>Object</code>)
<code>GetType()</code>	Gets the <code>Type</code> of the current instance. (Inherited from <code>Object</code>)
<code>MemberwiseClone()</code>	Creates a shallow copy of the current <code>Object</code> . (Inherited from <code>Object</code>)
<code>ToString()</code>	Returns a string that represents the current object. (Inherited from <code>Object</code>)

Explicit Interface Implementations

[] Expand table

<code>ICollection.CopyTo(Array, Int32)</code>	Copies the elements of the <code>ICollection</code> to an <code>Array</code> , starting at a particular <code>Array</code> index.
<code>ICollection.IsSynchronized</code>	Gets a value indicating whether access to the <code>ICollection</code> is synchronized (thread safe).
<code>ICollection.SyncRoot</code>	Gets an object that can be used to synchronize access to the <code>ICollection</code> .
<code>ICollection< TValue >.Add(TValue)</code>	Adds an item to the <code>ICollection< T ></code> . This implementation always throws <code>NotSupportedException</code> .
<code>ICollection< TValue >.Clear()</code>	Removes all items from the <code>ICollection< T ></code> . This implementation always throws <code>NotSupportedException</code> .
<code>ICollection< TValue >.Contains(TValue)</code>	Determines whether the <code>ICollection< T ></code> contains a specific value.
<code>ICollection< TValue >.IsReadOnly</code>	Gets a value indicating whether the <code>ICollection< T ></code> is read-only.
<code>ICollection< TValue >.Remove(TValue)</code>	Removes the first occurrence of a specific object from the <code>ICollection< T ></code> . This implementation always throws <code>NotSupportedException</code> .
<code>IEnumerable.GetEnumerator()</code>	Returns an enumerator that iterates through a collection.
<code>IEnumerable< TValue >.Get Enumerator()</code>	Returns an enumerator that iterates through a collection.

Extension Methods

[] Expand table

<code>ToImmutableArray< TSource >(IEnumerable< TSource >)</code>	Creates an immutable array from the specified collection.
<code>ToImmutableDictionary< TSource, TKey >(IEnumerable< TSource >, Func< TSource, TKey >, IEqualityComparer< TKey >)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary< TSource, TKey >(IEnumerable< TSource >, Func< TSource, TKey >)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.

<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its

	contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)	Applies an accumulator function over a sequence.
Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether all elements of a sequence satisfy a condition.
Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether any element of a sequence satisfies a condition.
Any<TSource>(IEnumerable<TSource>)	Determines whether a sequence contains any elements.
Append<TSource>(IEnumerable<TSource>, TSource)	Appends a value to the end of the sequence.
AsEnumerable<TSource>(IEnumerable<TSource>)	Returns the input typed as

	<code>IEnumerable<T>.</code>
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the average of a sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified

	<code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource,TKey>(IQueryable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IQueryable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IQueryable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IQueryable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IQueryable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IQueryable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IQueryable<TSource>, IQueryable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IQueryable<TSource>, IQueryable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IQueryable<TSource>, IQueryable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IQueryable<TSource>, IQueryable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified

	<p>condition.</p>
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a

	<p>result value from each group and its key. The keys are compared by using a specified comparer.</p>
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	<p>Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	<p>Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	<p>Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	<p>Produces the set intersection of two sequences by using the default equality comparer to compare values.</p>

<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a

	sequence.
Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Returns the maximum value in a generic sequence.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the maximum Double value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the maximum Single value.

<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Decimal</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Double</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Int32</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Int64</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable <code>Decimal</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable <code>Double</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.

<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEqualityComparer<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing

	their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then

	returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is

	true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.

ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.

InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`Shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [Dictionary<TKey,TValue>.ValueCollection](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Dictionary<TKey,TValue>.ValueCollection Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [Dictionary<TKey,TValue>.ValueCollection](#) class that reflects the values in the specified [Dictionary<TKey,TValue>](#).

C#

```
public ValueCollection(System.Collections.Generic.Dictionary<TKey, TValue>
dictionary);
```

Parameters

dictionary [Dictionary<TKey,TValue>](#)

The [Dictionary<TKey,TValue>](#) whose values are reflected in the new [Dictionary<TKey,TValue>.ValueCollection](#).

Exceptions

[ArgumentNullException](#)

dictionary is `null`.

Remarks

The [Dictionary<TKey,TValue>.ValueCollection](#) is not a static copy; instead, the [Dictionary<TKey,TValue>.ValueCollection](#) refers back to the values in the original [Dictionary<TKey,TValue>](#). Therefore, changes to the [Dictionary<TKey,TValue>](#) continue to be reflected in the [Dictionary<TKey,TValue>.ValueCollection](#).

This constructor is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.ValueCollection.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements contained in the [Dictionary<TKey,TValue>.ValueCollection](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [Dictionary<TKey,TValue>.ValueCollection](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Remarks

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >.ValueCollection. CopyTo(TValue[], Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [Dictionary< TKey, TValue >.ValueCollection](#) elements to an existing one-dimensional [Array](#), starting at the specified array index.

C#

```
public void CopyTo(TValue[] array, int index);
```

Parameters

array [TValue\[\]](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [Dictionary< TKey, TValue >.ValueCollection](#). The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

The number of elements in the source [Dictionary< TKey, TValue >.ValueCollection](#) is greater than the available space from [index](#) to the end of the destination [array](#).

Remarks

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [Dictionary<TKey,TValue>.ValueCollection](#).

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Array](#)

Dictionary<TKey,TValue>.ValueCollection.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [Dictionary<TKey,TValue>.ValueCollection](#).

C#

```
public  
System.Collections.Generic.Dictionary<TKey,TValue>.ValueCollection.Enumerator  
GetEnumerator();
```

Returns

[Dictionary<TKey,TValue>.ValueCollection.Enumerator](#)

A [Dictionary<TKey,TValue>.ValueCollection.Enumerator](#) for the [Dictionary<TKey,TValue>.ValueCollection](#).

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, `Current` is undefined. You must call `MoveNext` to advance the enumerator to the first element of the collection before reading the value of `Current`.

The `Current` returns the same object until `MoveNext` is called. `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this

position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [IEnumerator<T>](#)

Dictionary<TKey, TValue>.ValueCollection. ICollection<TValue>.Add Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to the [ICollection<T>](#). This implementation always throws [NotSupportedException](#).

C#

```
void ICollection<TValue>.Add(TValue item);
```

Parameters

item TValue

The object to add to the [ICollection<T>](#).

Implements

[Add\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

Dictionary< TKey, TValue >.ValueCollection. ICollection< TValue >.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all items from the [ICollection< T >](#). This implementation always throws [NotSupportedException](#).

C#

```
void ICollection<TValue>.Clear();
```

Implements

[Clear\(\)](#)

Exceptions

[NotSupportedException](#)

Always thrown.

Remarks

[Count](#) is set to zero, and references to other objects from elements of the collection are also released.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

Dictionary< TKey, TValue >.ValueCollection. ICollection< TValue >.Contains Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [ICollection<T>](#) contains a specific value.

C#

```
bool ICollection<TValue>.Contains(TValue item);
```

Parameters

item TValue

The object to locate in the [ICollection<T>](#).

Returns

Boolean

`true` if `item` is found in the [ICollection<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas, [Dictionary< TKey, TValue >](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.ValueCollection. ICollection<TValue>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool System.Collections.Generic.ICollection<TValue>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [Dictionary<TKey,TValue>.ValueCollection](#), this property always returns `true`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

Dictionary<TKey,TValue>.ValueCollection. ICollection<TValue>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a specific object from the [ICollection<T>](#). This implementation always throws [NotSupportedException](#).

C#

```
bool ICollection<TValue>.Remove(TValue item);
```

Parameters

item TValue

The object to remove from the [ICollection<T>](#).

Returns

Boolean

`true` if `item` was successfully removed from the [ICollection<T>](#); otherwise, `false`. This method also returns `false` if `item` was not found in the original [ICollection<T>](#).

Implements

[Remove\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown.

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas, [Dictionary<TKey,TValue>](#) allows the user to specify the [IComparer<T>](#)

implementation to use for comparing keys.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

Dictionary< TKey, TValue >.ValueCollection. IEnumerable< TValue >.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerator<TValue>
IEnumerable<TValue>.GetEnumerator();
```

Returns

[IEnumerator< TValue >](#)

An [IEnumerator< T >](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

`Current` returns the same object until `MoveNext` is called. `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this position, subsequent calls to `MoveNext` also return `false`. If the last call to `MoveNext` returned `false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to `MoveNext` or `IEnumerator.Reset` throws an `InvalidOperationException`.

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are `Remove` and `Clear`.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in `System.Collections.Generic` are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

Dictionary< TKey, TValue >.ValueCollection. ICollection.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the non-generic implementations of [ICollection.CopyTo](#) throw [InvalidOperationException](#), whereas the generic implementations throw [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)
- [Array](#)

Dictionary< TKey, TValue >.ValueCollection. ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [Dictionary< TKey, TValue >.ValueCollection](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object, which can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock this object before accessing the collection.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

Dictionary< TKey, TValue >.ValueCollection. ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [Dictionary< TKey, TValue >.ValueCollection](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object, which can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock this object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;  
lock (ic.SyncRoot) {  
    // Access the collection.  
}
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

Dictionary< TKey, TValue >.ValueCollection. IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator](#)

Dictionary<TKey,TValue> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a collection of keys and values.

C#

```
public class Dictionary<TKey, TValue> :  
    System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>, System.Collections.Generic.IDictionary<TKey, TValue>,  
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>>,  
    System.Collections.Generic.IReadOnlyCollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>, System.Collections.Generic.IReadOnlyDictionary<TKey, TValue>,  
    System.Collections.IDictionary,  
    System.Runtime.Serialization.IDeserializationCallback,  
    System.Runtime.Serialization.ISerializable
```

Type Parameters

TKey

The type of the keys in the dictionary.

TValue

The type of the values in the dictionary.

Inheritance [Object](#) → Dictionary<TKey,TValue>

Derived [System.ServiceModel.MessageQuerySet](#)

Implements [ICollection<KeyValuePair<TKey,TValue>>](#) , [IDictionary<TKey,TValue>](#) ,
[IEnumerable<KeyValuePair<TKey,TValue>>](#) , [IEnumerable<T>](#) ,
[IReadOnlyCollection<KeyValuePair<TKey,TValue>>](#) ,
[IReadOnlyDictionary<TKey,TValue>](#) , [ICollection](#) , [IDictionary](#) , [IEnumerable](#) ,
[IDeserializationCallback](#) , [ISerializable](#)

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.

```
C#  
  
// 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.  
*/
```

Remarks

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.

(!) Note

The speed of retrieval depends on the quality of the hashing algorithm of the type specified for `TKey`.

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<T>](#) generic interface by using a constructor that accepts a `comparer` parameter; if you do not specify an implementation, the default generic equality comparer [EqualityComparer<T>.Default](#) is used. If type `TKey` implements the [System.IEquatable<T>](#) generic interface, the default equality comparer uses that implementation.

(!) Note

For example, you can use the case-insensitive string comparers provided by the [StringComparer](#) class to create dictionaries with case-insensitive string keys.

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 `<gcAllowVeryLargeObjects>` 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.

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 the `Dictionary<TKey,TValue>` is a collection of keys and values, the element type is not the type of the key or the type of the value. Instead, the element type is a `KeyValuePair<TKey,TValue>` of the key type and the value type. For example:

C#

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

The `foreach` statement is a wrapper around the enumerator, which allows only reading from the collection, not writing to it.

 **Note**

Because keys can be inherited and their behavior changed, their absolute uniqueness cannot be guaranteed by comparisons using the `Equals` method.

Constructors

 Expand table

<code>Dictionary<TKey,TValue>()</code>	Initializes a new instance of the <code>Dictionary<TKey,TValue></code> class that is empty, has the default initial capacity, and uses the default equality comparer for the key type.
<code>Dictionary<TKey,TValue>(IDictionary<TKey,TValue>, IEqualityComparer<TKey>)</code>	Initializes a new instance of the <code>Dictionary<TKey,TValue></code> class that contains elements copied from the specified <code>IDictionary<TKey,TValue></code> and uses the specified <code>IEqualityComparer<T></code> .
<code>Dictionary<TKey,TValue>(IDictionary<TKey,TValue>)</code>	Initializes a new instance of the <code>Dictionary<TKey,TValue></code> class that contains elements copied from the specified <code>IDictionary<TKey,TValue></code> and uses the default equality comparer for the key type.

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

Properties

[+] Expand table

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

Methods

[+] Expand table

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)	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)
OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(TKey, TValue)	Removes the value with the specified key from the Dictionary<TKey,TValue> , and copies the element to the value parameter.
Remove(TKey)	Removes the value with the specified key from the Dictionary<TKey,TValue> .
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.

Explicit Interface Implementations

 [Expand table](#)

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection<T> to an array, starting at the specified array index.
ICollection.IsSynchronized	Gets a value that indicates whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
ICollection<KeyValuePair< TKey, TValue >>.Add(KeyValuePair< TKey, TValue >)	Adds the specified value to the ICollection<T> with the specified key.
ICollection<KeyValuePair< TKey, TValue >>.Contains(KeyValuePair< TKey, TValue >)	Determines whether the ICollection<T> contains a specific key and value.
ICollection<KeyValuePair< TKey, TValue >>.CopyTo(KeyValuePair< TKey, TValue >[], Int32)	Copies the elements of the ICollection<T> to an array of type KeyValuePair< TKey, TValue > , starting at the specified array index.
ICollection<KeyValuePair< TKey, TValue >>.IsReadOnly	Gets a value that indicates whether the dictionary is read-only.
ICollection<KeyValuePair< TKey, TValue >>.Remove(KeyValuePair< TKey, TValue >)	Removes a key and value from the dictionary.
IDictionary.Add(Object, Object)	Adds the specified key and value to the dictionary.
IDictionary.Contains(Object)	Determines whether the IDictionary contains an element with the specified key.
IDictionary.GetEnumerator()	Returns an IDictionaryEnumerator for the IDictionary .
IDictionary.IsFixedSize	Gets a value that indicates whether the IDictionary has a fixed size.
IDictionary.IsReadOnly	Gets a value that indicates whether the IDictionary is read-only.

IDictionary.Item[Object]	Gets or sets the value with the specified key.
IDictionary.Keys	Gets an ICollection containing the keys of the IDictionary .
IDictionary.Remove(Object)	Removes the element with the specified key from the IDictionary .
IDictionary.Values	Gets an ICollection containing the values in the IDictionary .
IDictionary< TKey, TValue >.Keys	Gets an ICollection< T > containing the keys of the IDictionary< TKey, TValue > .
IDictionary< TKey, TValue >.Values	Gets an ICollection< T > containing the values in the IDictionary< TKey, TValue > .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through the collection.
IEnumerable< KeyValuePair< TKey, TValue > >.Get Enumerator()	Returns an enumerator that iterates through the collection.
IReadOnlyDictionary< TKey, TValue >.Keys	Gets a collection containing the keys of the IReadOnlyDictionary< TKey, TValue > .
IReadOnlyDictionary< TKey, TValue >.Values	Gets a collection containing the values of the IReadOnlyDictionary< TKey, TValue > .

Extension Methods

[Expand table](#)

GetValueOrDefault< TKey, TValue >(IReadOnly Dictionary< TKey, TValue >, TKey, TValue)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
GetValueOrDefault< TKey, TValue >(IReadOnly Dictionary< TKey, TValue >, TKey)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
Remove< TKey, TValue >(IDictionary< TKey, TValue >, TKey, TValue)	Tries to remove the value with the specified <code>key</code> from the <code>dictionary</code> .
TryAdd< TKey, TValue >(IDictionary< TKey, TValue >, TKey, TValue)	Tries to add the specified <code>key</code> and <code>value</code> to the <code>dictionary</code> .
ToImmutableArray< TSource >(IEnumerable< TSource >)	Creates an immutable array from the specified collection.

<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its

	contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.

<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.

<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>)</code>	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.

<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key

<code>Comparer<TKey></code>	selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>,</code>	Correlates the elements of two sequences based on equality of

<code>Func<TOuter, IEnumerable<TInner>, TResult>()</code>	keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.

<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.

<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a

	<p>result selector function on each element therein.</p>
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	<p>Projects each element of a sequence to an IEnumerable<T>, flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	<p>Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T>.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	<p>Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.</p>
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.</p>
Single<TSource>(IEnumerable<TSource>)	<p>Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	<p>Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	<p>Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified</p>

	condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform

	function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Computes the sum of the sequence of nullable <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Computes the sum of the sequence of <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.

<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToList<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .

ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an IEnumerable to an IQueryable .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of every node in the source collection.
<code>DescendantNodes<T>(IEnumerable<T>)</code>	Returns a collection of the descendant nodes of every document and element in the source collection.
<code>Descendants<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
<code>Descendants<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the descendant

	elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

A [Dictionary<TKey,TValue>](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. In the rare case where an enumeration contends with write accesses, the collection must be locked during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

For thread-safe alternatives, see the [ConcurrentDictionary<TKey,TValue>](#) class or [ImmutableDictionary<TKey,TValue>](#) class.

Public static (`Shared` in Visual Basic) members of this type are thread safe.

See also

- [IDictionary<TKey,TValue>](#)
- [SortedList<TKey,TValue>](#)
- [KeyValuePair<TKey,TValue>](#)
- [IEqualityComparer<T>](#)

Dictionary<TKey,TValue> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [Dictionary<TKey,TValue>](#) class.

Overloads

[+] [Expand table](#)

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>(IEnumerable<KeyValuePair<TKey,TValue>>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IEnumerable<T> .
Dictionary<TKey,TValue>(IEqualityComparer<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>(IDictionary<TKey,TValue>, IEqualityComparer<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<KeyValuePair<TKey,TValue>>, IEqualityComparer<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> .

<code>IEqualityComparer<TKey>)</code>	elements copied from the specified <code>IEnumerable<T></code> and uses the specified <code>IEqualityComparer<T></code> .
<code>Dictionary<TKey,TValue>(Int32, IEqualityComparer<TKey>)</code>	Initializes a new instance of the <code>Dictionary<TKey,TValue></code> class that is empty, has the specified initial capacity, and uses the specified <code>IEqualityComparer<T></code> .
<code>Dictionary<TKey,TValue>(SerializationInfo, StreamingContext)</code>	Initializes a new instance of the <code>Dictionary<TKey,TValue></code> class with serialized data.

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.

C#

```
public Dictionary();
```

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.

This code example is part of a larger example provided for the `Dictionary<TKey,TValue>` class.

C#

```
// 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.");
}
```

Remarks

Every key in a [Dictionary<TKey,TValue>](#) must be unique according to the default equality comparer.

[Dictionary<TKey,TValue>](#) requires an equality implementation to determine whether keys are equal. This constructor uses the default generic equality comparer, [EqualityComparer<T>.Default](#). If type [TKey](#) implements the [System.IEquatable<T>](#) generic interface, the default equality comparer uses that implementation. Alternatively, you can specify an implementation of the [IEqualityComparer<T>](#) generic interface by using a constructor that accepts a [comparer](#) parameter.

ⓘ Note

If you can estimate the size of the collection, using a constructor that specifies the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Dictionary<TKey,TValue>](#).

This constructor is an O(1) operation.

See also

- [IImmutableDictionary<TKey,TValue>](#)
- [Default](#)
- [IEquatable<T>](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public Dictionary(System.Collections.Generic.IDictionary<TKey, TValue>
dictionary);
```

Parameters

dictionary [IDictionary<TKey,TValue>](#)

The [IDictionary<TKey,TValue>](#) whose elements are copied to the new [Dictionary<TKey,TValue>](#).

Exceptions

[ArgumentNullException](#)

`dictionary` is `null`.

[ArgumentException](#)

`dictionary` contains one or more duplicate keys.

Examples

The following code example shows how to use the [Dictionary<TKey,TValue>\(IEqualityComparer<TKey>\)](#) constructor to initialize a [Dictionary<TKey,TValue>](#) with sorted content from another dictionary. The code example creates a

`SortedDictionary< TKey, TValue >` and populates it with data in random order, then passes the `SortedDictionary< TKey, TValue >` to the `Dictionary< TKey, TValue >` (`IEqualityComparer< TKey >`) constructor, creating a `Dictionary< TKey, TValue >` that is sorted. This is useful if you need to build a sorted dictionary that at some point becomes static; copying the data from a `SortedDictionary< TKey, TValue >` to a `Dictionary< TKey, TValue >` improves retrieval speed.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string
        // keys.
        SortedDictionary<string, string> openWith =
            new SortedDictionary<string, string>();

        // Add some elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Create a Dictionary of strings with string keys, and
        // initialize it with the contents of the sorted dictionary.
        Dictionary<string, string> copy =
            new Dictionary<string, string>(openWith);

        // List the contents of the copy.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in copy )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }

    /* This code example produces the following output:
     * 
     * Key = bmp, Value = paint.exe
     * Key = dib, Value = paint.exe
     * Key = rtf, Value = wordpad.exe
     * Key = txt, Value = notepad.exe
     */
}
```

Remarks

Every key in a [Dictionary<TKey,TValue>](#) must be unique according to the default equality comparer; likewise, every key in the source `dictionary` must also be unique according to the default equality comparer.

The initial capacity of the new [Dictionary<TKey,TValue>](#) is large enough to contain all the elements in `dictionary`.

[Dictionary<TKey,TValue>](#) requires an equality implementation to determine whether keys are equal. This constructor uses the default generic equality comparer, [EqualityComparer<T>.Default](#). If type `TKey` implements the [System.IEquatable<T>](#) generic interface, the default equality comparer uses that implementation. Alternatively, you can specify an implementation of the [IEqualityComparer<T>](#) generic interface by using a constructor that accepts a `comparer` parameter.

This constructor is an O(n) operation, where n is the number of elements in `dictionary`.

See also

- [IDictionary<TKey,TValue>](#)
- [Default](#)
- [IEquatable<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue> ([IEnumerable<KeyValuePair<TKey,TValue>>](#))

Initializes a new instance of the [Dictionary<TKey,TValue>](#) class that contains elements copied from the specified [IEnumerable<T>](#).

C#

```
public Dictionary(System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>> collection);
```

Parameters

collection [IEnumerable<KeyValuePair<TKey,TValue>>](#)

The [IEnumerable<T>](#) whose elements are copied to the new [Dictionary<TKey,TValue>](#).

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

[ArgumentException](#)

`collection` contains one or more duplicated keys.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Dictionary<TKey,TValue> ([IEqualityComparer<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>](#).

C#

```
public Dictionary(System.Collections.Generic.IEqualityComparer<TKey>?
```

```
comparer);
```

Parameters

comparer [IEqualityComparer<TKey>](#)

The [IEqualityComparer<T>](#) implementation to use when comparing keys, or `null` to use the default [EqualityComparer<T>](#) for the type of the key.

Examples

The following code example creates a [Dictionary<TKey,TValue>](#) with a case-insensitive equality comparer for the current culture. The example adds four elements, some with lower-case keys and some with upper-case keys. The example then attempts to add an element with a key that differs from an existing key only by case, catches the resulting exception, and displays an error message. Finally, the example displays the elements in the dictionary.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new Dictionary of strings, with string keys
        // and a case-insensitive comparer for the current culture.
        Dictionary<string, string> openWith =
            new Dictionary<string, string>(
                StringComparer.CurrentCultureIgnoreCase);

        // Add some elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("DIB", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Try to add a fifth element with a key that is the same
        // except for case; this would be allowed with the default
        // comparer.
        try
        {
            openWith.Add("BMP", "paint.exe");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("\nBMP is already in the dictionary.");
        }
    }
}
```

```

    }

    // List the contents of the sorted dictionary.
    Console.WriteLine();
    foreach( KeyValuePair<string, string> kvp in openWith )
    {
        Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
                          kvp.Value);
    }
}

/* This code example produces the following output:

BMP is already in the dictionary.

Key = txt, Value = notepad.exe
Key = bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
*/

```

Remarks

Use this constructor with the case-insensitive string comparers provided by the [StringComparer](#) class to create dictionaries with case-insensitive string keys.

Every key in a [Dictionary<TKey,TValue>](#) must be unique according to the specified comparer.

[Dictionary<TKey,TValue>](#) requires an equality implementation to determine whether keys are equal. If `comparer` is `null`, this constructor uses the default generic equality comparer, [EqualityComparer<T>.Default](#). If type `TKey` implements the [System.IEquatable<T>](#) generic interface, the default equality comparer uses that implementation.

ⓘ Note

If you can estimate the size of the collection, using a constructor that specifies the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Dictionary<TKey,TValue>](#).

This constructor is an O(1) operation.

See also

- [IEqualityComparer<T>](#)
- [Default](#)
- [IEquatable<T>](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public Dictionary(int capacity);
```

Parameters

capacity Int32

The initial number of elements that the [Dictionary<TKey,TValue>](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#)

`capacity` is less than 0.

Examples

The following code example creates a dictionary with an initial capacity of 4 and populates it with 4 entries.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys and
        // an initial capacity of 4.
        Dictionary<string, string> openWith =
            new Dictionary<string, string>(4);

        // Add 4 elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // List the contents of the dictionary.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }
}

/* This code example produces the following output:

Key = txt, Value = notepad.exe
Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = rtf, Value = wordpad.exe
*/
```

Remarks

Every key in a [Dictionary<TKey,TValue>](#) must be unique according to the default equality comparer.

The capacity of a [Dictionary<TKey,TValue>](#) is the number of elements that can be added to the [Dictionary<TKey,TValue>](#) before resizing is necessary. As elements are added to a [Dictionary<TKey,TValue>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the

Dictionary<TKey,TValue>.

Dictionary<TKey,TValue> requires an equality implementation to determine whether keys are equal. This constructor uses the default generic equality comparer, EqualityComparer<T>.Default. If type TKey implements the System.IEquatable<T> generic interface, the default equality comparer uses that implementation. Alternatively, you can specify an implementation of the IEqualityComparer<T> generic interface by using a constructor that accepts a comparer parameter.

This constructor is an O(1) operation.

See also

- [Default](#)
- [IEquatable<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue> (IDictionary<TKey,TValue>, IEqualityComparer<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>.

C#

```
public Dictionary(System.Collections.Generic.IDictionary<TKey,TValue>
dictionary, System.Collections.Generic.IEqualityComparer<TKey>? comparer);
```

Parameters

dictionary [IDictionary<TKey,TValue>](#)

The [IDictionary<TKey,TValue>](#) whose elements are copied to the new [Dictionary<TKey,TValue>](#).

comparer [IEqualityComparer<TKey>](#)

The [IEqualityComparer<T>](#) implementation to use when comparing keys, or `null` to use the default [EqualityComparer<T>](#) for the type of the key.

Exceptions

[ArgumentNullException](#)

`dictionary` is `null`.

[ArgumentException](#)

`dictionary` contains one or more duplicate keys.

Examples

The following code example shows how to use the [Dictionary<TKey,TValue>](#) ([IDictionary<TKey,TValue>](#), [IEqualityComparer<TKey>](#)) constructor to initialize a [Dictionary<TKey,TValue>](#) with case-insensitive sorted content from another dictionary. The code example creates a [SortedDictionary<TKey,TValue>](#) with a case-insensitive comparer and populates it with data in random order, then passes the [SortedDictionary<TKey,TValue>](#) to the [Dictionary<TKey,TValue>\(IDictionary<TKey,TValue>, IEqualityComparer<TKey>\)](#) constructor, along with a case-insensitive equality comparer, creating a [Dictionary<TKey,TValue>](#) that is sorted. This is useful if you need to build a sorted dictionary that at some point becomes static; copying the data from a [SortedDictionary<TKey,TValue>](#) to a [Dictionary<TKey,TValue>](#) improves retrieval speed.

Note

When you create a new dictionary with a case-insensitive comparer and populate it with entries from a dictionary that uses a case-sensitive comparer, as in this example, an exception occurs if the input dictionary has keys that differ only by case.

C#

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

```

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string
        // keys and a case-insensitive comparer.
        SortedDictionary<string, string> openWith =
            new SortedDictionary<string, string>(
                StringComparer.CurrentCultureIgnoreCase);

        // Add some elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("Bmp", "paint.exe");
        openWith.Add("DIB", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Create a Dictionary of strings with string keys and a
        // case-insensitive equality comparer, and initialize it
        // with the contents of the sorted dictionary.
        Dictionary<string, string> copy =
            new Dictionary<string, string>(openWith,
                StringComparer.CurrentCultureIgnoreCase);

        // List the contents of the copy.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in copy )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }
}

/* This code example produces the following output:

Key = Bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/

```

Remarks

Use this constructor with the case-insensitive string comparers provided by the [StringComparer](#) class to create dictionaries with case-insensitive string keys.

Every key in a [Dictionary<TKey,TValue>](#) must be unique according to the specified comparer; likewise, every key in the source `dictionary` must also be unique according to the specified comparer.

ⓘ Note

For example, duplicate keys can occur if `comparer` is one of the case-insensitive string comparers provided by the [StringComparer](#) class and `dictionary` does not use a case-insensitive comparer key.

The initial capacity of the new [Dictionary<TKey,TValue>](#) is large enough to contain all the elements in `dictionary`.

[Dictionary<TKey,TValue>](#) requires an equality implementation to determine whether keys are equal. If `comparer` is `null`, this constructor uses the default generic equality comparer, [EqualityComparer<T>.Default](#). If type `TKey` implements the [System.IEquatable<T>](#) generic interface, the default equality comparer uses that implementation.

This constructor is an $O(n)$ operation, where `n` is the number of elements in `dictionary`.

See also

- [IDictionary<TKey,TValue>](#)
- [IEqualityComparer<T>](#)
- [Default](#)
- [IEquatable<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue> ([IEnumerable<KeyValuePair<TKey,TValue>>](#), [IEqualityComparer<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>](#).

C#

```
public  
Dictionary<System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>> collection,  
System.Collections.Generic.IEqualityComparer<TKey>? comparer);
```

Parameters

collection [IEnumerable<KeyValuePair<TKey,TValue>>](#)

The [IEnumerable<T>](#) whose elements are copied to the new [Dictionary<TKey,TValue>](#).

comparer [IEqualityComparer<TKey>](#)

The [IEqualityComparer<T>](#) implementation to use when comparing keys, or `null` to use the default [EqualityComparer<T>](#) for the type of the key.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

[ArgumentException](#)

`collection` contains one or more duplicated keys.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Dictionary<TKey,TValue>(Int32, IEqualityComparer<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>](#).

C#

```
public Dictionary(int capacity,  
System.Collections.Generic.IEqualityComparer<TKey>? comparer);
```

Parameters

capacity [Int32](#)

The initial number of elements that the [Dictionary<TKey,TValue>](#) can contain.

comparer [IEqualityComparer<TKey>](#)

The [IEqualityComparer<T>](#) implementation to use when comparing keys, or [null](#) to use the default [EqualityComparer<T>](#) for the type of the key.

Exceptions

[ArgumentOutOfRangeException](#)

`capacity` is less than 0.

Examples

The following code example creates a [Dictionary<TKey,TValue>](#) with an initial capacity of 5 and a case-insensitive equality comparer for the current culture. The example adds four elements, some with lower-case keys and some with upper-case keys. The example then attempts to add an element with a key that differs from an existing key only by case, catches the resulting exception, and displays an error message. Finally, the example displays the elements in the dictionary.

C#

```
using System;  
using System.Collections.Generic;  
  
public class Example  
{  
    public static void Main()  
    {  
        // Create a new dictionary of strings, with string keys, an  
        // initial capacity of 5, and a case-insensitive equality  
        // comparer.  
        Dictionary<string, string> openWith =
```

```

        new Dictionary<string, string>(5,
            StringComparer.CurrentCultureIgnoreCase);

    // Add 4 elements to the dictionary.
    openWith.Add("txt", "notepad.exe");
    openWith.Add("bmp", "paint.exe");
    openWith.Add("DIB", "paint.exe");
    openWith.Add("rtf", "wordpad.exe");

    // Try to add a fifth element with a key that is the same
    // except for case; this would be allowed with the default
    // comparer.
    try
    {
        openWith.Add("BMP", "paint.exe");
    }
    catch (ArgumentException)
    {
        Console.WriteLine("\nBMP is already in the dictionary.");
    }

    // List the contents of the dictionary.
    Console.WriteLine();
    foreach( KeyValuePair<string, string> kvp in openWith )
    {
        Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
            kvp.Value);
    }
}

/* This code example produces the following output:

BMP is already in the dictionary.

Key = txt, Value = notepad.exe
Key = bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
*/

```

Remarks

Use this constructor with the case-insensitive string comparers provided by the [StringComparer](#) class to create dictionaries with case-insensitive string keys.

Every key in a [Dictionary<TKey,TValue>](#) must be unique according to the specified comparer.

The capacity of a [Dictionary<TKey,TValue>](#) is the number of elements that can be added to the [Dictionary<TKey,TValue>](#) before resizing is necessary. As elements are added to a

`Dictionary< TKey, TValue >`, the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the `Dictionary< TKey, TValue >`.

`Dictionary< TKey, TValue >` requires an equality implementation to determine whether keys are equal. If `comparer` is `null`, this constructor uses the default generic equality comparer, `EqualityComparer< T >.Default`. If type `TKey` implements the `System.IEquatable< T >` generic interface, the default equality comparer uses that implementation.

This constructor is an O(1) operation.

See also

- [IEqualityComparer< T >](#)
- [Default](#)
- [IEquatable< T >](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue > (SerializationInfo, StreamingContext)

Initializes a new instance of the `Dictionary< TKey, TValue >` class with serialized data.

C#

```
protected Dictionary(System.Runtime.Serialization.SerializationInfo info,
```

```
System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object containing the information required to serialize the [Dictionary<TKey,TValue>](#).

context [StreamingContext](#)

A [StreamingContext](#) structure containing the source and destination of the serialized stream associated with the [Dictionary<TKey,TValue>](#).

Remarks

This constructor is called during deserialization to reconstitute an object transmitted over a stream. For more information, see [XML and SOAP Serialization](#).

See also

- [System.Runtime.Serialization](#)

Applies to

▼ .NET 10 and other versions

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

Dictionary< TKey, TValue >.Comparer Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the [IEqualityComparer<T>](#) that is used to determine equality of keys for the dictionary.

C#

```
public System.Collections.Generic.IEqualityComparer< TKey > Comparer { get; }
```

Property Value

[IEqualityComparer< TKey >](#)

The [IEqualityComparer<T>](#) generic interface implementation that is used to determine equality of keys for the current [Dictionary< TKey, TValue >](#) and to provide hash values for the keys.

Remarks

[Dictionary< TKey, TValue >](#) requires an equality implementation to determine whether keys are equal. You can specify an implementation of the [IEqualityComparer< T >](#) generic interface by using a constructor that accepts a `comparer` parameter; if you do not specify one, the default generic equality comparer [EqualityComparer< T >.Default](#) is used.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of key/value pairs contained in the [Dictionary< TKey, TValue >](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of key/value pairs contained in the [Dictionary< TKey, TValue >](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Remarks

The capacity of a [Dictionary< TKey, TValue >](#) is the number of elements that the [Dictionary< TKey, TValue >](#) can store. The [Count](#) property is the number of elements that are actually in the [Dictionary< TKey, TValue >](#).

The capacity is always greater than or equal to [Count](#). If [Count](#) exceeds the capacity while adding elements, the capacity is increased by automatically reallocating the internal array before copying the old elements and adding the new elements.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.Item[TKey] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the value associated with the specified key.

C#

```
public TValue this[TKey key] { get; set; }
```

Parameters

key TKey

The key of the value to get or set.

Property Value

TValue

The value associated with the specified key. If the specified key is not found, a get operation throws a [KeyNotFoundException](#), and a set operation creates a new element with the specified key.

Implements

[Item\[TKey\]](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[KeyNotFoundException](#)

The property is retrieved and `key` does not exist in the collection.

Examples

The following code 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 also 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.

This code example is part of a larger example provided for the [Dictionary<TKey,TValue>](#) class. `openWith` is the name of the Dictionary used in this example.

C#

```
// 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.");  
}
```

C#

```
// 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";
```

C#

```
// 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.");  
}
```

C#

```
// 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.");  
}
```

Remarks

This property provides the ability to access a specific element in the collection by using the following C# syntax: `myCollection[key]` (`myCollection(key)` in Visual Basic).

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the [Dictionary<TKey,TValue>](#). When you set the property value, if the key is in the [Dictionary<TKey,TValue>](#), the value associated with that key is replaced by the assigned value. If the key is not in the [Dictionary<TKey,TValue>](#), the key and value are added to the dictionary. In contrast, the [Add](#) method does not modify existing elements.

A key cannot be `null`, but a value can be, if the value type `TValue` is a reference type.

The C# language uses the [this](#) keyword to define the indexers instead of implementing the [Item\[\]](#) property. Visual Basic implements [Item\[\]](#) as a default property, which provides the same indexing functionality.

Getting or setting the value of this property approaches an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Add\(TKey, TValue\)](#)

Dictionary< TKey, TValue >.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the keys in the [Dictionary< TKey, TValue >](#).

C#

```
public System.Collections.Generic.Dictionary< TKey, TValue >.KeyCollection Keys {  
    get; }
```

Property Value

[Dictionary< TKey, TValue >.KeyCollection](#)

A [Dictionary< TKey, TValue >.KeyCollection](#) containing the keys in the [Dictionary< TKey, TValue >](#).

Examples

The following code example shows how to enumerate the keys in the dictionary using the [Keys](#) property, and how to enumerate the keys and values in the dictionary.

This code is part of a larger example that can be compiled and executed (`openWith` is the name of the Dictionary used in this example). See [Dictionary< TKey, TValue >](#).

C#

```
// 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);  
}
```

C#

```
// 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);
}
```

Remarks

The order of the keys in the [Dictionary<TKey,TValue>.KeyCollection](#) is unspecified, but it is the same order as the associated values in the [Dictionary<TKey,TValue>.ValueCollection](#) returned by the [Values](#) property.

The returned [Dictionary<TKey,TValue>.KeyCollection](#) is not a static copy; instead, the [Dictionary<TKey,TValue>.KeyCollection](#) refers back to the keys in the original [Dictionary<TKey,TValue>](#). Therefore, changes to the [Dictionary<TKey,TValue>](#) continue to be reflected in the [Dictionary<TKey,TValue>.KeyCollection](#).

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.KeyCollection](#)
- [Values](#)

Dictionary< TKey, TValue >.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the values in the [Dictionary< TKey, TValue >](#).

C#

```
public System.Collections.Generic.Dictionary< TKey, TValue >.ValueCollection Values {  
    get; }
```

Property Value

[Dictionary< TKey, TValue >.ValueCollection](#)

A [Dictionary< TKey, TValue >.ValueCollection](#) containing the values in the [Dictionary< TKey, TValue >](#).

Examples

This code example shows how to enumerate the values in the dictionary using the [Values](#) property, and how to enumerate the keys and values in the dictionary.

This code example is part of a larger example provided for the [Dictionary< TKey, TValue >](#) class (`openWith` is the name of the Dictionary used in this example).

C#

```
// 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);  
}
```

C#

```
// 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);
}
```

Remarks

The order of the values in the [Dictionary<TKey,TValue>.ValueCollection](#) is unspecified, but it is the same order as the associated keys in the [Dictionary<TKey,TValue>.KeyCollection](#) returned by the [Keys](#) property.

The returned [Dictionary<TKey,TValue>.ValueCollection](#) is not a static copy; instead, the [Dictionary<TKey,TValue>.ValueCollection](#) refers back to the values in the original [Dictionary<TKey,TValue>](#). Therefore, changes to the [Dictionary<TKey,TValue>](#) continue to be reflected in the [Dictionary<TKey,TValue>.ValueCollection](#).

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.ValueCollection](#)
- [Keys](#)

Dictionary<TKey,TValue>.Add(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds the specified key and value to the dictionary.

C#

```
public void Add(TKey key, TValue value);
```

Parameters

key TKey

The key of the element to add.

value TValue

The value of the element to add. The value can be `null` for reference types.

Implements

[Add\(TKey, TValue\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

An element with the same key already exists in the [Dictionary<TKey,TValue>](#).

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.

This code example is part of a larger example provided for the [Dictionary<TKey,TValue>](#) class.

C#

```
// 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.");  
}
```

Remarks

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the [Dictionary<TKey,TValue>](#); for example, `myCollection[myKey] = myValue` (in Visual Basic, `myCollection(myKey) = myValue`). However, if the specified key already exists in the [Dictionary<TKey,TValue>](#), setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method throws an exception if a value with the specified key already exists.

If the [Count](#) property value already equals the capacity, the capacity of the [Dictionary<TKey,TValue>](#) is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

A key cannot be `null`, but a value can be, if `Tvalue` is a reference type.

If [Count](#) is less than the capacity, this method approaches an O(1) operation. If the capacity must be increased to accommodate the new element, this method becomes an O(`n`) operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove\(TKey\)](#)
- [Item\[TKey\]](#)
- [Add\(TKey, TValue\)](#)

Dictionary<TKey,TValue>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all keys and values from the [Dictionary<TKey,TValue>](#).

C#

```
public void Clear();
```

Implements

[Clear\(\)](#) , [Clear\(\)](#)

Remarks

The [Count](#) property is set to 0, and references to other objects from elements of the collection are also released. The capacity remains unchanged.

This method is an O(n) operation, where n is the capacity of the dictionary.

.NET Core 3.0+ only: this mutating method may be safely called without invalidating active enumerators on the [Dictionary<TKey,TValue>](#) instance. This does not imply thread safety.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.ContainsKey(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [Dictionary<TKey,TValue>](#) contains the specified key.

C#

```
public bool ContainsKey(TKey key);
```

Parameters

key TKey

The key to locate in the [Dictionary<TKey,TValue>](#).

Returns

Boolean

`true` if the [Dictionary<TKey,TValue>](#) contains an element with the specified key; otherwise, `false`.

Implements

[ContainsKey\(TKey\)](#) , [ContainsKey\(TKey\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [ContainsKey](#) method to test whether a key exists prior to calling the [Add](#) method. It also shows how to use the [TryGetValue](#) method to retrieve values, which is an efficient way to retrieve values when a program frequently tries keys

that are not in the dictionary. Finally, it shows the least efficient way to test whether keys exist, by using the [Item\[\]](#) property (the indexer in C#).

This code example is part of a larger example provided for the [Dictionary<TKey,TValue>](#) class (`openWith` is the name of the Dictionary used in this example).

C#

```
// 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"]);
}
```

C#

```
// 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.");
}
```

C#

```
// 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.");
}
```

Remarks

This method approaches an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsValue\(TValue\)](#)

Dictionary< TKey, TValue >.ContainsValue(TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [Dictionary< TKey, TValue >](#) contains a specific value.

C#

```
public bool ContainsValue(TValue value);
```

Parameters

value TValue

The value to locate in the [Dictionary< TKey, TValue >](#). The value can be `null` for reference types.

Returns

Boolean

`true` if the [Dictionary< TKey, TValue >](#) contains an element with the specified value; otherwise, `false`.

Remarks

This method determines equality using the default equality comparer [EqualityComparer< T >.Default](#) for `TValue`, the type of values in the dictionary.

This method performs a linear search; therefore, the average execution time is proportional to [Count](#). That is, this method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [Default](#)

Dictionary< TKey, TValue >.EnsureCapacity(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Ensures that the dictionary can hold up to a specified number of entries without any further expansion of its backing storage.

C#

```
public int EnsureCapacity(int capacity);
```

Parameters

capacity [Int32](#)

The number of entries.

Returns

[Int32](#)

The current capacity of the [Dictionary< TKey, TValue >](#).

Exceptions

[ArgumentOutOfRangeException](#)

capacity is less than 0.

Applies to

Product	Versions
.NET	Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Dictionary< TKey, TValue >.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [Dictionary< TKey, TValue >](#).

C#

```
public System.Collections.Generic.Dictionary< TKey, TValue >.Enumerator  
GetEnumerator();
```

Returns

[Dictionary< TKey, TValue >.Enumerator](#)

A [Dictionary< TKey, TValue >.Enumerator](#) structure for the [Dictionary< TKey, TValue >](#).

Remarks

For purposes of enumeration, each item is a [KeyValuePair< TKey, TValue >](#) structure representing a value and its key.

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same element until the [MoveNext](#) method is called.

[MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this

position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dictionary<TKey,TValue>.Enumerator](#)
- [IEnumerator<T>](#)

Dictionary<TKey,TValue>.GetObjectData Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and returns the data needed to serialize the Dictionary<TKey,TValue> instance.

C#

```
public virtual void GetObjectData(System.Runtime.Serialization.SerializationInfo  
info, System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information required to serialize the Dictionary<TKey,TValue> instance.

context [StreamingContext](#)

A [StreamingContext](#) structure that contains the source and destination of the serialized stream associated with the Dictionary<TKey,TValue> instance.

Implements

[GetObjectData\(SerializationInfo, StreamingContext\)](#)

Exceptions

[ArgumentNullException](#)

`info` is `null`.

Remarks

This method is an O(`n`) operation, where `n` is [Count](#).

Applies to

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

See also

- [ISerializable](#)
- [SerializationInfo](#)
- [StreamingContext](#)
- [OnDeserialization\(Object\)](#)

Dictionary< TKey, TValue >.OnDeserialization(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and raises the deserialization event when the deserialization is complete.

C#

```
public virtual void OnDeserialization(object? sender);
```

Parameters

sender [Object](#)

The source of the deserialization event.

Implements

[OnDeserialization\(Object\)](#)

Exceptions

[SerializationException](#)

The [SerializationInfo](#) object associated with the current [Dictionary< TKey, TValue >](#) instance is invalid.

Remarks

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

See also

- [ISerializable](#)
- [SerializationInfo](#)
- [StreamingContext](#)
- [GetObjectData\(SerializationInfo, StreamingContext\)](#)

Dictionary<TKey,TValue>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Overloads

 [Expand table](#)

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 <code>value</code> parameter.

Remove(TKey)

Removes the value with the specified key from the [Dictionary<TKey,TValue>](#).

C#

```
public bool Remove(TKey key);
```

Parameters

key TKey

The key of the element to remove.

Returns

Boolean

`true` if the element is successfully found and removed; otherwise, `false`. This method returns `false` if `key` is not found in the [Dictionary<TKey,TValue>](#).

Implements

[Remove\(TKey\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to remove a key/value pair from a dictionary using the `Remove` method.

This code example is part of a larger example provided for the [Dictionary<TKey,TValue>](#) class (`openWith` is the name of the Dictionary used in this example).

C#

```
// 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.");  
}
```

Remarks

If the [Dictionary<TKey,TValue>](#) does not contain an element with the specified key, the [Dictionary<TKey,TValue>](#) remains unchanged. No exception is thrown.

This method approaches an O(1) operation.

.NET Core 3.0+ only: this mutating method may be safely called without invalidating active enumerators on the [Dictionary<TKey,TValue>](#) instance. This does not imply thread safety.

See also

- [Add\(TKey, TValue\)](#)
- [Remove\(TKey\)](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Remove(TKey, TValue)

Removes the value with the specified key from the [Dictionary<TKey,TValue>](#), and copies the element to the `value` parameter.

C#

```
public bool Remove(TKey key, out TValue value);
```

Parameters

key TKey

The key of the element to remove.

value TValue

The removed element.

Returns

[Boolean](#)

`true` if the element is successfully found and removed; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Dictionary< TKey, TValue >.TrimExcess Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Overloads

 [Expand table](#)

TrimExcess(Int32)	Sets the capacity of this dictionary to hold up a specified number of entries without any further expansion of its backing storage.
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.

C#

```
public void TrimExcess(int capacity);
```

Parameters

capacity Int32

The new capacity.

Exceptions

[ArgumentOutOfRangeException](#)

capacity is less than [Count](#).

Remarks

This method can be used to minimize the memory overhead once it is known that no new elements will be added.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

TrimExcess()

Sets the capacity of this dictionary to what it would be if it had been originally initialized with all its entries.

C#

```
public void TrimExcess();
```

Remarks

This method can be used to minimize memory overhead once it is known that no new elements will be added to the dictionary. To allocate a minimum size storage array, execute the following statements:

C#

```
dictionary.Clear();
dictionary.TrimExcess();
```

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Dictionary<TKey,TValue>.TryAdd(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Attempts to add the specified key and value to the dictionary.

C#

```
public bool TryAdd(TKey key, TValue value);
```

Parameters

key TKey

The key of the element to add.

value TValue

The value of the element to add. It can be `null`.

Returns

Boolean

`true` if the key/value pair was added to the dictionary successfully; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Remarks

Unlike the [Add](#) method, this method doesn't throw an exception if the element with the given key exists in the dictionary. Unlike the Dictionary indexer, `TryAdd` doesn't override the element if the element with the given key exists in the dictionary. If the key already exists, `TryAdd` does nothing and returns `false`.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Dictionary<TKey,TValue>.TryGetValue(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the value associated with the specified key.

C#

```
public bool TryGetValue(TKey key, out TValue value);
```

Parameters

key TKey

The key of the value to get.

value TValue

When this method returns, contains the value associated with the specified key, if the key is found; otherwise, the default value for the type of the **value** parameter. This parameter is passed uninitialized.

Returns

Boolean

true if the [Dictionary<TKey,TValue>](#) contains an element with the specified key; otherwise, **false**.

Implements

[TryGetValue\(TKey, TValue\)](#) , [TryGetValue\(TKey, TValue\)](#)

Exceptions

[ArgumentNullException](#)

key is **null**.

Examples

The example shows how to use the [TryGetValue](#) method as a more efficient way to retrieve values in a program that frequently tries keys that are not in the dictionary. For contrast, the example also shows how the [Item\[\]](#) property (the indexer in C#) throws exceptions when attempting to retrieve nonexistent keys.

This code example is part of a larger example provided for the [Dictionary<TKey,TValue>](#) class (openWith is the name of the Dictionary used in this example).

C#

```
// 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.");
}
```

C#

```
// 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.");
}
```

Remarks

This method combines the functionality of the [ContainsKey](#) method and the [Item\[\]](#) property.

If the key is not found, then the [value](#) parameter gets the appropriate default value for the type [TValue](#); for example, 0 (zero) for integer types, [false](#) for Boolean types, and [null](#) for reference types.

Use the [TryGetValue](#) method if your code frequently attempts to access keys that are not in the dictionary. Using this method is more efficient than catching the [KeyNotFoundException](#) thrown by the [Item\[\]](#) property.

This method approaches an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [Item\[TKey\]](#)

Dictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.Add Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds the specified value to the [ICollection<T>](#) with the specified key.

C#

```
void  
ICollection<KeyValuePair< TKey, TValue >>.Add(System.Collections.Generic.KeyValuePair  
< TKey, TValue > keyValuePair);
```

Parameters

keyValuePair [KeyValuePair< TKey, TValue >](#)

The [KeyValuePair< TKey, TValue >](#) structure representing the key and value to add to the [Dictionary< TKey, TValue >](#).

Implements

[Add\(T\)](#)

Exceptions

[ArgumentNullException](#)

The key of `keyValuePair` is `null`.

[ArgumentException](#)

An element with the same key already exists in the [Dictionary< TKey, TValue >](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.Contains Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [ICollection<T>](#) contains a specific key and value.

C#

```
bool  
ICollection<KeyValuePair< TKey , TValue >>.Contains(System.Collections.Generic.KeyValue  
ePair< TKey , TValue > keyValuePair);
```

Parameters

keyValuePair [KeyValuePair< TKey, TValue >](#)

The [KeyValuePair< TKey, TValue >](#) structure to locate in the [ICollection< T >](#).

Returns

Boolean

`true` if `keyValuePair` is found in the [ICollection< T >](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection<T>](#) to an array of type [KeyValuePair< TKey, TValue >](#), starting at the specified array index.

C#

```
void  
ICollection<KeyValuePair< TKey, TValue >>.CopyTo(System.Collections.Generic.KeyValueP  
air< TKey, TValue >[] array, int index);
```

Parameters

array [KeyValuePair< TKey, TValue > \[\]](#)

The one-dimensional array of type [KeyValuePair< TKey, TValue >](#) that is the destination of the [KeyValuePair< TKey, TValue >](#) elements copied from the [ICollection< T >](#). The array must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than 0.

[ArgumentException](#)

The number of elements in the source `ICollection<T>` is greater than the available space from `index` to the end of the destination `array`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value that indicates whether the dictionary is read-only.

C#

```
bool  
System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [Dictionary< TKey, TValue >](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes a key and value from the dictionary.

C#

```
bool  
ICollection<KeyValuePair< TKey, TValue >>.Remove(System.Collections.Generic.KeyValuePair< TKey, TValue > keyValuePair);
```

Parameters

keyValuePair [KeyValuePair< TKey, TValue >](#)

The [KeyValuePair< TKey, TValue >](#) structure representing the key and value to remove from the [Dictionary< TKey, TValue >](#).

Returns

[Boolean](#)

`true` if the key and value represented by `keyValuePair` is successfully found and removed; otherwise, `false`. This method returns `false` if `keyValuePair` is not found in the [ICollection< T >](#).

Implements

[Remove\(T\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.IDictionary<TKey,TValue>.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection<T>](#) containing the keys of the [IDictionary<TKey,TValue>](#).

C#

```
System.Collections.Generic.ICollection<TKey>
System.Collections.Generic.IDictionary<TKey, TValue>.Keys { get; }
```

Property Value

[ICollection<TKey>](#)

An [ICollection<T>](#) of type [TKey](#) containing the keys of the [IDictionary<TKey,TValue>](#).

Implements

[Keys](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey, TValue>.IDictionary<TKey, TValue>.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection<T>](#) containing the values in the [IDictionary<TKey,TValue>](#).

C#

```
System.Collections.Generic.ICollection<TValue>
System.Collections.Generic.IDictionary<TKey, TValue>.Values { get; }
```

Property Value

[ICollection<TValue>](#)

An [ICollection<T>](#) of type [TValue](#) containing the values in the [IDictionary<TKey,TValue>](#).

Implements

[Values](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.Generic.IEnumerator<System.Collections.Generic.KeyValuePair<TKey, TValue>> IEnumerable<KeyValuePair<TKey, TValue>>.GetEnumerator();
```

Returns

[IEnumerator<KeyValuePair<TKey,TValue>>](#)

An enumerator that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >. IReadOnlyDictionary< TKey, TValue >.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the keys of the [IReadOnlyDictionary< TKey, TValue >](#).

C#

```
System.Collections.Generic.IEnumerable< TKey >
System.Collections.Generic.IReadOnlyDictionary< TKey, TValue >.Keys { get; }
```

Property Value

[IEnumerable< TKey >](#)

A collection containing the keys of the [IReadOnlyDictionary< TKey, TValue >](#).

Implements

[Keys](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.IReadOnlyDictionary<TKey,TValue>.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the values of the [IReadOnlyDictionary<TKey,TValue>](#).

C#

```
System.Collections.Generic.IEnumerable<TValue>
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>.Values { get; }
```

Property Value

[IEnumerable<TValue>](#)

A collection containing the values of the [IReadOnlyDictionary<TKey,TValue>](#).

Implements

[Values](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey, TValue>.ICollection.CopyTo To(Array, Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection<T>](#) to an array, starting at the specified array index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional array that is the destination of the elements copied from [ICollection<T>](#).
The array must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than 0.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection<T>](#) is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source [ICollection<T>](#) cannot be cast automatically to the type of the destination `array`.

Remarks

Each element copied from a [Dictionary<TKey,TValue>](#) is a [KeyValuePair<TKey,TValue>](#) structure representing a value and its key.

! [Note](#)

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the nongeneric implementations of [ICollection.CopyTo](#) throw an [InvalidCastException](#), whereas the generic implementations throw an [ArgumentException](#).

This method is an O(`n`) operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value that indicates whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [Dictionary< TKey, TValue >](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. Even when a collection is synchronized, other threads can still modify the collection, which can cause the enumerator to throw an exception. To guarantee thread safety during enumeration, you can either lock the collection during the entire enumeration or catch the exceptions resulting from changes made by other threads.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

Dictionary<TKey,TValue>.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#).

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;  
lock (ic.SyncRoot) {
```

```
// Access the collection.  
}
```

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

Dictionary<TKey,TValue>.IDictionary.Add(Object, Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds the specified key and value to the dictionary.

C#

```
void IDictionary.Add(object key, object value);
```

Parameters

key [Object](#)

The object to use as the key.

value [Object](#)

The object to use as the value.

Implements

[Add\(Object, Object\)](#)

Exceptions

[ArgumentNullException](#)

key is `null`.

[ArgumentException](#)

key is of a type that is not assignable to the key type `TKey` of the [Dictionary<TKey,TValue>](#).

-or-

value is of a type that is not assignable to `TValue`, the type of values in the [Dictionary<TKey,TValue>](#).

-or-

A value with the same key already exists in the `Dictionary<TKey,TValue>`.

Examples

The following code example shows how to access the `Dictionary<TKey,TValue>` class through the `System.Collections.IDictionary` interface. The code example creates an empty `Dictionary<TKey,TValue>` of strings with string keys and uses the `IDictionary.Add` method to add some elements. The example demonstrates that the `IDictionary.Add` method throws an `ArgumentException` when attempting to add a duplicate key, or when a key or value of the wrong data type is supplied.

The code example demonstrates the use of several other members of the `System.Collections.IDictionary` interface.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //

        IDictionary openWith = new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
        try
        {
            openWith.Add(42, new Example());
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("An exception was caught for " +
                "IDictionary.Add. Exception message:\n\t{0}\n",
                ex.Message);
        }

        // 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 returns null if the key is of the wrong data
// type.
Console.WriteLine("The indexer returns null"
    + " if the key is of the wrong type:");
Console.WriteLine("For key = 2, value = {0}.",
    openWith[2]);

// The indexer throws an exception when setting a value
// if the key is of the wrong data type.
try
{
    openWith[2] = "This does not get added.";
}
catch (ArgumentException)
{
    Console.WriteLine("A key of the wrong type was specified"
        + " when assigning to the indexer.");
}

// Unlike the default Item property on the Dictionary class
// itself, IDictionary.Item does not throw an exception
// if the requested key is not in the dictionary.
Console.WriteLine("For key = \"tif\", value = {0}.",
    openWith["tif"]);

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

```
        openWith["ht"]);
    }

    // IDictionary.Contains returns false if the wrong data
    // type is supplied.
    Console.WriteLine("openWith.Contains(29.7) returns {0}",
        openWith.Contains(29.7));

    // When you use foreach to enumerate dictionary elements
    // with the IDictionary interface, the elements are retrieved
    // as DictionaryEntry objects instead of 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 icoll = openWith.Values;

    // The elements of the collection are strongly typed
    // with the type that was specified for dictionary values,
    // even though the ICollection interface is not strongly
    // typed.
    Console.WriteLine();
    foreach( string s in icoll )
    {
        Console.WriteLine("Value = {0}", s);
    }

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

    // The elements of the collection are strongly typed
    // with the type that was specified for dictionary keys,
    // even though the ICollection interface is not strongly
    // typed.
    Console.WriteLine();
    foreach( string s in icoll )
    {
        Console.WriteLine("Key = {0}", s);
    }

    // Use the Remove method to remove a key/value pair. No
    // exception is thrown if the wrong data type is supplied.
    Console.WriteLine("\nRemove(\"dib\")");
    openWith.Remove("dib");

    if (!openWith.Contains("dib"))
    {
        Console.WriteLine("Key \"dib\" is not found.");
    }
}
```

```

/* This code example produces the following output:

An exception was caught for IDictionary.Add. Exception message:
    The value "42" is not of type "System.String" and cannot be used in this
generic collection.
Parameter name: key

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
The indexer returns null if the key is of the wrong type:
For key = 2, value = .
A key of the wrong type was specified when assigning to the indexer.
For key = "tif", value = .
Value added for key = "ht": hypertrm.exe
openWith.Contains(29.7) returns False

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("dib")
Key "dib" is not found.
*/

```

Remarks

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the dictionary, setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method throws an exception if the specified key already exists.

If [Count](#) is less than the capacity, this method approaches an O(1) operation. If the capacity needs to be increased to accommodate the new element, this method becomes an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Item\[Object\]](#)

Dictionary< TKey, TValue >.IDictionary.Contains(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [IDictionary](#) contains an element with the specified key.

C#

```
bool IDictionary.Contains(object key);
```

Parameters

key [Object](#)

The key to locate in the [IDictionary](#).

Returns

[Boolean](#)

`true` if the [IDictionary](#) contains an element with the specified key; otherwise, `false`.

Implements

[Contains\(Object\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [IDictionary.Contains](#) method of the [System.Collections.IDictionary](#) interface with a [Dictionary< TKey, TValue >](#). The example demonstrates that the method returns `false` if a key of the wrong data type is supplied.

The code example is part of a larger example, including output, provided for the `IDictionary.Add` method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
```

C#

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

// IDictionary.Contains returns false if the wrong data
// type is supplied.
Console.WriteLine("openWith.Contains(29.7) returns {0}",
    openWith.Contains(29.7));
```

C#

```
}
```

Remarks

This method returns `false` if `key` is of a type that is not assignable to the key type `TKey` of the [Dictionary<TKey,TValue>](#).

This method approaches an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.IDictionary.Get Enumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an [IDictionaryEnumerator](#) for the [IDictionary](#).

C#

```
System.Collections.IDictionaryEnumerator IDictionary.GetEnumerator();
```

Returns

[IDictionaryEnumerator](#)

An [IDictionaryEnumerator](#) for the [IDictionary](#).

Implements

[GetEnumerator\(\)](#)

Examples

The following code example shows how to enumerate the key/value pairs in the dictionary by using the `foreach` statement (`For Each` in Visual Basic), which hides the use of the enumerator. In particular, note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair<TKey,TValue>](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
{
```

```
public static void Main()
{
    // Create a new dictionary of strings, with string keys,
    // and access it using the IDictionary interface.
    //
    IDictionary openWith = new Dictionary<string, string>();

    // Add some elements to the dictionary. There are no
    // duplicate keys, but some of the values are duplicates.
    // IDictionary.Add throws an exception if incorrect types
    // are supplied for key or value.
    openWith.Add("txt", "notepad.exe");
    openWith.Add("bmp", "paint.exe");
    openWith.Add("dib", "paint.exe");
    openWith.Add("rtf", "wordpad.exe");
}
```

C#

```
// When you use foreach to enumerate dictionary elements
// with the IDictionary interface, the elements are retrieved
// as DictionaryEntry objects instead of KeyValuePair objects.
Console.WriteLine();
foreach( DictionaryEntry de in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
        de.Key, de.Value);
}
```

C#

```
}
```

Remarks

For purposes of enumeration, each item is a [DictionaryEntry](#) structure.

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. The [Reset](#) method also brings the enumerator back to this position. At this position, [Entry](#) is undefined.

Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Entry](#).

The [Entry](#) property returns the same element until either the [MoveNext](#) or [Reset](#) method is called. [MoveNext](#) sets [Entry](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Entry](#) is undefined. To set [Entry](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [IEnumerator](#)

Dictionary<TKey,TValue>.IDictionary.IsFixedSize Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value that indicates whether the [IDictionary](#) has a fixed size.

C#

```
bool System.Collections.IDictionary.IsFixedSize { get; }
```

Property Value

[Boolean](#)

`true` if the [IDictionary](#) has a fixed size; otherwise, `false`. In the default implementation of [Dictionary<TKey,TValue>](#), this property always returns `false`.

Implements

[IsFixedSize](#)

Remarks

A collection with a fixed size does not allow the addition or removal of elements after the collection is created, but it allows the modification of existing elements.

A collection with a fixed size is simply a collection with a wrapper that prevents adding and removing elements; therefore, if changes are made to the underlying collection, including the addition or removal of elements, the fixed-size collection reflects those changes.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.IDictionary.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value that indicates whether the [IDictionary](#) is read-only.

C#

```
bool System.Collections.IDictionary.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [IDictionary](#) is read-only; otherwise, `false`. In the default implementation of [Dictionary<TKey,TValue>](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary<TKey,TValue>.IDictionary. Item[Object] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the value with the specified key.

C#

```
object? System.Collections.IDictionary.Item[object key] { get; set; }
```

Parameters

key [Object](#)

The key of the value to get.

Property Value

[Object](#)

The value associated with the specified key, or `null` if `key` is not in the dictionary or `key` is of a type that is not assignable to the key type `TKey` of the [Dictionary<TKey,TValue>](#).

Implements

[Item\[Object\]](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

A value is being assigned, and `key` is of a type that is not assignable to the key type `TKey` of the [Dictionary<TKey,TValue>](#).

-or-

A value is being assigned and is of a type that isn't assignable to the value type `TValue` of the `Dictionary<TKey,TValue>`.

Examples

The following code example shows how to use the `IDictionary.Item[]` property (the indexer in C#) of the `System.Collections.IDictionary` interface with a `Dictionary<TKey,TValue>`, and ways the property differs from the `Dictionary<TKey,TValue>.Item[]` property.

The example shows that, like the `Dictionary<TKey,TValue>.Item[]` property, the `Dictionary<TKey,TValue>.IDictionary.Item[]` property can change the value associated with an existing key and can be used to add a new key/value pair if the specified key is not in the dictionary. The example also shows that unlike the `Dictionary<TKey,TValue>.Item[]` property, the `Dictionary<TKey,TValue>.IDictionary.Item[]` property does not throw an exception if `key` is not in the dictionary, returning a null reference instead. Finally, the example demonstrates that getting the `Dictionary<TKey,TValue>.IDictionary.Item[]` property returns a null reference if `key` is not the correct data type, and that setting the property throws an exception if `key` is not the correct data type.

The code example is part of a larger example, including output, provided for the `IDictionary.Add` method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
```

C#

```
// 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 returns null if the key is of the wrong data
// type.
Console.WriteLine("The indexer returns null"
    + " if the key is of the wrong type:");
Console.WriteLine("For key = 2, value = {0}.",
    openWith[2]);

// The indexer throws an exception when setting a value
// if the key is of the wrong data type.
try
{
    openWith[2] = "This does not get added.";
}
catch (ArgumentException)
{
    Console.WriteLine("A key of the wrong type was specified"
        + " when assigning to the indexer.");
}
```

C#

```
// Unlike the default Item property on the Dictionary class
// itself, IDictionary.Item does not throw an exception
// if the requested key is not in the dictionary.
Console.WriteLine("For key = \"tif\", value = {0}.",
    openWith["tif"]);
```

C#

```
}
```

Remarks

This property provides the ability to access a specific value in the collection by using the following C# syntax: `myCollection[key]` (`myCollection(key)` in Visual Basic).

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the dictionary, setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

The C# language uses the [this](#) keyword to define the indexers instead of implementing the [IDictionary.Item\[\]](#) property. Visual Basic implements [IDictionary.Item\[\]](#) as a default property, which provides the same indexing functionality.

Getting or setting the value of this property approaches an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Add\(Object, Object\)](#)

Dictionary< TKey, TValue >.IDictionary.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection](#) containing the keys of the [IDictionary](#).

C#

```
System.Collections.ICollection System.Collections.IDictionary.Keys { get; }
```

Property Value

[ICollection](#)

An [ICollection](#) containing the keys of the [IDictionary](#).

Implements

[Keys](#)

Examples

The following code example shows how to use the [IDictionary.Keys](#) property of the [System.Collections.IDictionary](#) interface with a [Dictionary< TKey, TValue >](#), to list the keys in the dictionary. The example also shows how to enumerate the key/value pairs in the dictionary; note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
```

```
{  
    public static void Main()  
    {  
        // Create a new dictionary of strings, with string keys,  
        // and access it using the IDictionary interface.  
        //  
        IDictionary openWith = new Dictionary<string, string>();  
  
        // Add some elements to the dictionary. There are no  
        // duplicate keys, but some of the values are duplicates.  
        // IDictionary.Add throws an exception if incorrect types  
        // are supplied for key or value.  
        openWith.Add("txt", "notepad.exe");  
        openWith.Add("bmp", "paint.exe");  
        openWith.Add("dib", "paint.exe");  
        openWith.Add("rtf", "wordpad.exe");
```

C#

```
// To get the keys alone, use the Keys property.  
icoll = openWith.Keys;  
  
// The elements of the collection are strongly typed  
// with the type that was specified for dictionary keys,  
// even though the ICollection interface is not strongly  
// typed.  
Console.WriteLine();  
foreach( string s in icoll )  
{  
    Console.WriteLine("Key = {0}", s);  
}
```

C#

```
// When you use foreach to enumerate dictionary elements  
// with the IDictionary interface, the elements are retrieved  
// as DictionaryEntry objects instead of KeyValuePair objects.  
Console.WriteLine();  
foreach( DictionaryEntry de in openWith )  
{  
    Console.WriteLine("Key = {0}, Value = {1}",  
        de.Key, de.Value);  
}
```

C#

```
}  
}
```

Remarks

The order of the keys in the returned [ICollection](#) is unspecified, but it is guaranteed to be the same order as the corresponding values in the [ICollection](#) returned by the [Values](#) property.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)

Dictionary< TKey, TValue >.IDictionary.Remove(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the element with the specified key from the [IDictionary](#).

C#

```
void IDictionary.Remove(object key);
```

Parameters

key [Object](#)

The key of the element to remove.

Implements

[Remove\(Object\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [IDictionary.Remove](#) of the [System.Collections.IDictionary](#) interface with a [Dictionary< TKey, TValue >](#).

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

```

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
    }
}

```

C#

```

// Use the Remove method to remove a key/value pair. No
// exception is thrown if the wrong data type is supplied.
Console.WriteLine("\nRemove(\"dib\")");
openWith.Remove("dib");

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

```

C#

```

}
}
```

Remarks

This method approaches an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Dictionary< TKey, TValue >. IDictionary. Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection](#) containing the values in the [IDictionary](#).

C#

```
System.Collections.ICollection System.Collections.IDictionary.Values { get; }
```

Property Value

[ICollection](#)

An [ICollection](#) containing the values in the [IDictionary](#).

Implements

[Values](#)

Examples

The following code example shows how to use the [IDictionary.Values](#) property of the [System.Collections.IDictionary](#) interface with a [Dictionary< TKey, TValue >](#), to list the values in the dictionary. The example also shows how to enumerate the key/value pairs in the dictionary; note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
```

```
{  
    public static void Main()  
    {  
        // Create a new dictionary of strings, with string keys,  
        // and access it using the IDictionary interface.  
        //  
        IDictionary openWith = new Dictionary<string, string>();  
  
        // Add some elements to the dictionary. There are no  
        // duplicate keys, but some of the values are duplicates.  
        // IDictionary.Add throws an exception if incorrect types  
        // are supplied for key or value.  
        openWith.Add("txt", "notepad.exe");  
        openWith.Add("bmp", "paint.exe");  
        openWith.Add("dib", "paint.exe");  
        openWith.Add("rtf", "wordpad.exe");
```

C#

```
// To get the values alone, use the Values property.  
ICollection icoll = openWith.Values;  
  
// The elements of the collection are strongly typed  
// with the type that was specified for dictionary values,  
// even though the ICollection interface is not strongly  
// typed.  
Console.WriteLine();  
foreach( string s in icoll )  
{  
    Console.WriteLine("Value = {0}", s);  
}
```

C#

```
// When you use foreach to enumerate dictionary elements  
// with the IDictionary interface, the elements are retrieved  
// as DictionaryEntry objects instead of KeyValuePair objects.  
Console.WriteLine();  
foreach( DictionaryEntry de in openWith )  
{  
    Console.WriteLine("Key = {0}, Value = {1}",  
        de.Key, de.Value);  
}
```

C#

```
}  
}
```

Remarks

The order of the values in the returned [ICollection](#) is unspecified, but it is guaranteed to be the same order as the corresponding keys in the [ICollection](#) returned by the [Keys](#) property.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)

Dictionary<TKey, TValue>.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. The [Reset](#) method also brings the enumerator back to this position. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same element until either the [MoveNext](#) or [Reset](#) method is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding elements or changing the capacity, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

.NET Core 3.0+ only: The only mutating methods which do not invalidate enumerators are [Remove](#) and [Clear](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator](#)

EqualityComparer<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Provides a base class for implementations of the [IEqualityComparer<T>](#) generic interface.

C#

```
public abstract class EqualityComparer<T> :  
    System.Collections.Generic.IEqualityComparer<T>,  
    System.Collections.IEqualityComparer
```

Type Parameters

T

The type of objects to compare.

Inheritance [Object](#) → EqualityComparer<T>

Implements [IEqualityComparer<T>](#) , [IEqualityComparer](#)

Examples

The following example creates a dictionary collection of objects of type [Box](#) with an equality comparer. Two boxes are considered equal if their dimensions are the same. It then adds the boxes to the collection.

The dictionary is recreated with an equality comparer that defines equality in a different way: Two boxes are considered equal if their volumes are the same.

C#

```
using System;  
using System.Collections.Generic;  
  
class Program  
{  
    static Dictionary<Box, String> boxes;  
  
    static void Main()
```

```

{
    BoxSameDimensions boxDim = new BoxSameDimensions();
    boxes = new Dictionary<Box, string>(boxDim);

    Console.WriteLine("Boxes equality by dimensions:");
    Box redBox = new Box(8, 4, 8);
    Box greenBox = new Box(8, 6, 8);
    Box blueBox = new Box(8, 4, 8);
    Box yellowBox = new Box(8, 8, 8);
    AddBox(redBox, "red");
    AddBox(greenBox, "green");
    AddBox(blueBox, "blue");
    AddBox(yellowBox, "yellow");

    Console.WriteLine();
    Console.WriteLine("Boxes equality by volume:");

    BoxSameVolume boxVolume = new BoxSameVolume();
    boxes = new Dictionary<Box, string>(boxVolume);
    Box pinkBox = new Box(8, 4, 8);
    Box orangeBox = new Box(8, 6, 8);
    Box purpleBox = new Box(4, 8, 8);
    Box brownBox = new Box(8, 8, 4);
    AddBox(pinkBox, "pink");
    AddBox(orangeBox, "orange");
    AddBox(purpleBox, "purple");
    AddBox(brownBox, "brown");
}

public static void AddBox(Box bx, string name)
{
    try
    {
        boxes.Add(bx, name);
        Console.WriteLine("Added {0}, Count = {1}, GetHashCode = {2}",
            name, boxes.Count.ToString(), bx.GetHashCode());
    }
    catch (ArgumentException)
    {
        Console.WriteLine("A box equal to {0} is already in the collection.",
name);
    }
}

public class Box
{
    public Box(int h, int l, int w)
    {
        this.Height = h;
        this.Length = l;
        this.Width = w;
    }

    public int Height { get; set; }

    public int Length { get; set; }
}

```

```

    public int Width { get; set; }

}

class BoxSameDimensions : EqualityComparer<Box>
{
    public override bool Equals(Box b1, Box b2)
    {
        if (b1 == null && b2 == null)
            return true;
        else if (b1 == null || b2 == null)
            return false;

        return (b1.Height == b2.Height &&
                b1.Length == b2.Length &&
                b1.Width == b2.Width);
    }

    public override int GetHashCode(Box bx)
    {
        int hCode = bx.Height ^ bx.Length ^ bx.Width;
        return hCode.GetHashCode();
    }
}

class BoxSameVolume : EqualityComparer<Box>
{
    public override bool Equals(Box b1, Box b2)
    {
        if (b1 == null && b2 == null)
            return true;
        else if (b1 == null || b2 == null)
            return false;

        return (b1.Height * b1.Width * b1.Length ==
                b2.Height * b2.Width * b2.Length);
    }

    public override int GetHashCode(Box bx)
    {
        int hCode = bx.Height * bx.Length * bx.Width;
        return hCode.GetHashCode();
    }
}

/* This example produces an output similar to the following:
 *
    Boxes equality by dimensions:
    Added red, Count = 1, HashCode = 46104728
    Added green, Count = 2, HashCode = 12289376
    A box equal to blue is already in the collection.
    Added yellow, Count = 3, HashCode = 43495525

    Boxes equality by volume:
    Added pink, Count = 1, HashCode = 55915408
    Added orange, Count = 2, HashCode = 33476626
    A box equal to purple is already in the collection.

```

```
* A box equal to brown is already in the collection.  
*/
```

Remarks

Derive from this class to provide a custom implementation of the [IEqualityComparer<T>](#) generic interface for use with collection classes such as the [Dictionary< TKey, TValue >](#) generic class, or with methods such as [List< T >.Sort](#).

The [Default](#) property checks whether type `T` implements the [System.IEquatable<T>](#) generic interface and, if so, returns an [EqualityComparer<T>](#) that invokes the implementation of the [IEquatable<T>.Equals](#) method. Otherwise, it returns an [EqualityComparer<T>](#), as provided by `T`.

In .NET 8 and later versions, we recommend using the [EqualityComparer<T>.Create\(Func<T,T,Boolean>, Func<T,Int32>\)](#) method to create instances of this type.

Constructors

[+] [Expand table](#)

EqualityComparer<T>()	Initializes a new instance of the EqualityComparer<T> class.
---	--

Properties

[+] [Expand table](#)

Default	Returns a default equality comparer for the type specified by the generic argument.
-------------------------	---

Methods

[+] [Expand table](#)

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
Equals(T, T)	When overridden in a derived class, determines whether two objects of type <code>T</code> are equal.

GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetHashCode(T)	When overridden in a derived class, serves as a hash function for the specified object for hashing algorithms and data structures, such as a hash table.
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object . (Inherited from Object)
ToString()	Returns a string that represents the current object. (Inherited from Object)

Explicit Interface Implementations

 [Expand table](#)

IEqualityComparer.Equals(Object, Object)	Determines whether the specified objects are equal.
IEqualityComparer.GetHashCode(Object)	Returns a hash code for the specified object.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEqualityComparer<T>](#)
- [IEquatable<T>](#)

EqualityComparer<T> Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [EqualityComparer<T>](#) class.

C#

```
protected EqualityComparer();
```

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

EqualityComparer<T>.Default Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns a default equality comparer for the type specified by the generic argument.

C#

```
public static System.Collections.Generic.EqualityComparer<T> Default { get; }
```

Property Value

[EqualityComparer<T>](#)

The default instance of the [EqualityComparer<T>](#) class for type `T`.

Examples

The following example creates a collection that contains elements of the `Box` type and then searches it for a box matching another box by calling the `FindFirst` method, twice.

The first search does not specify any equality comparer, which means `FindFirst` uses [EqualityComparer<T>.Default](#) to determine equality of boxes. That in turn uses the implementation of the [IEquatable<T>.Equals](#) method in the `Box` class. Two boxes are considered equal if their dimensions are the same.

The second search specifies an equality comparer (`BoxEqVolume`) that defines equality by volume. Two boxes are considered equal if their volumes are the same.

C#

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

static class Program
{
    static void Main()
    {
        var redBox = new Box(8, 8, 4);
        var blueBox = new Box(6, 8, 4);
        var greenBox = new Box(4, 8, 8);
```

```

        var boxes = new[] { redBox, blueBox, greenBox };

        var boxToFind = new Box(4, 8, 8);

        var foundByDimension = boxes.FindFirst(boxToFind);

        Console.WriteLine($"Found box {foundByDimension} by dimension.");

        var foundByVolume = boxes.FindFirst(boxToFind, new BoxEqVolume());

        Console.WriteLine($"Found box {foundByVolume} by volume.");
    }
}

public static class CollectionExtensions
{
    public static T FindFirst<T>(
        this IEnumerable<T> collection, T itemToFind, IEqualityComparer<T>
comparer = null)
    {
        comparer = comparer ?? EqualityComparer<T>.Default;

        foreach (var item in collection)
        {
            if (comparer.Equals(item, itemToFind))
            {
                return item;
            }
        }

        throw new InvalidOperationException("No matching item found.");
    }
}

public class BoxEqVolume : EqualityComparer<Box>
{
    public override bool Equals(Box b1, Box b2)
    {
        if (object.ReferenceEquals(b1, b2))
            return true;

        if (b1 is null || b2 is null)
            return false;

        return b1.Volume == b2.Volume;
    }

    public override int GetHashCode(Box box) => box.Volume.GetHashCode();
}

public class Box : IEquatable<Box>
{
    public Box(int height, int length, int width)
    {

```

```

        this.Height = height;
        this.Length = length;
        this.Width = width;
    }

    public int Height { get; }
    public int Length { get; }
    public int Width { get; }

    public int Volume => Height * Length * Width;

    public bool Equals(Box other)
    {
        if (other is null)
            return false;

        return this.Height == other.Height && this.Length == other.Length
            && this.Width == other.Width;
    }

    public override bool Equals(object obj) => Equals(obj as Box);
    public override int GetHashCode() => (Height, Length, Width).GetHashCode();

    public override string ToString() => $"{Height} x {Length} x {Width}";
}

/* This example produces the following output:
 *
 * Found box 4 x 8 x 8 by dimension.
 * Found box 8 x 8 x 4 by volume.
 */

```

Remarks

The [Default](#) property checks whether type `T` implements the [System.IEquatable<T>](#) interface and, if so, returns an [EqualityComparer<T>](#) that uses that implementation. Otherwise, it returns an [EqualityComparer<T>](#) that uses the overrides of [Object.Equals](#) and [Object.GetHashCode](#) provided by `T`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [IEqualityComparer<T>](#)
- [IEquatable<T>](#)
- [Object](#)

EqualityComparer<T>.Equals(T, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

When overridden in a derived class, determines whether two objects of type `T` are equal.

C#

```
public abstract bool Equals(T? x, T? y);
```

Parameters

`x` `T`

The first object to compare.

`y` `T`

The second object to compare.

Returns

[Boolean](#)

`true` if the specified objects are equal; otherwise, `false`.

Implements

[Equals\(T, T\)](#)

Remarks

The [Equals](#) method is reflexive, symmetric, and transitive. That is, it returns `true` if used to compare an object with itself; `true` for two objects `x` and `y` if it is `true` for `y` and `x`; and `true` for two objects `x` and `z` if it is `true` for `x` and `y` and also `true` for `y` and `z`.

Notes to Implementers

Implementations are required to ensure that if the `Equals(T, T)` method returns `true` for two objects `x` and `y`, then the value returned by the `GetHashCode(T)` method for `x` must equal the value returned for `y`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

EqualityComparer<T>.GetHashCode(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

When overridden in a derived class, serves as a hash function for the specified object for hashing algorithms and data structures, such as a hash table.

C#

```
public abstract int GetHashCode(T obj);
```

Parameters

obj T

The object for which to get a hash code.

Returns

[Int32](#)

A hash code for the specified object.

Implements

[GetHashCode\(T\)](#)

Exceptions

[ArgumentNullException](#)

The type of `obj` is a reference type and `obj` is `null`.

Notes to Implementers

Implementations are required to ensure that if the [Equals\(T, T\)](#) method returns `true` for two objects `x` and `y`, then the value returned by the [GetHashCode\(T\)](#) method for `x` must equal the value returned for `y`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

EqualityComparer<T>.IEqualityComparer.Equals(Object, Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the specified objects are equal.

C#

```
bool IEqualityComparer.Equals(object x, object y);
```

Parameters

x [Object](#)

The first object to compare.

y [Object](#)

The second object to compare.

Returns

[Boolean](#)

`true` if the specified objects are equal; otherwise, `false`.

Implements

[EqualityComparer<T>.IEqualityComparer.Equals\(Object, Object\)](#)

Exceptions

[ArgumentException](#)

`x` or `y` is of a type that cannot be cast to type `T`.

Remarks

This method is a wrapper for the [Equals\(T, T\)](#) method, so `obj` must be cast to the type specified by the generic argument `T` of the current instance. If it cannot be cast to `T`, an [ArgumentException](#) is thrown.

Comparing `null` is allowed and does not generate an exception.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

EqualityComparer<T>.IEqualityComparer.GetHashCode(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns a hash code for the specified object.

C#

```
int IEqualityComparer.GetHashCode(object obj);
```

Parameters

obj [Object](#)

The [Object](#) for which a hash code is to be returned.

Returns

[Int32](#)

A hash code for the specified object.

Implements

[GetHashCode\(Object\)](#)

Exceptions

[ArgumentNullException](#)

The type of **obj** is a reference type and **obj** is **null**.

-or-

obj is of a type that cannot be cast to type **T**.

Remarks

This method is a wrapper for the [GetHashCode\(T\)](#) method, so `obj` must be a type that can be cast to the type specified by the generic type argument `T` of the current instance.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [HashSet<T>](#) object.

C#

```
public struct HashSet<T>.Enumerator : System.Collections.Generic.IEnumerator<T>
```

Type Parameters

T

Inheritance [Object](#) → [ValueType](#) → [HashSet<T>.Enumerator](#)

Implements [IEnumerator<T>](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned

`false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator object instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to `MoveNext` or `IEnumerator.Reset` throws an `InvalidOperationException`.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the `System.Collections.Generic` namespace are not synchronized.

Properties

[] [Expand table](#)

<code>Current</code>	Gets the element at the current position of the enumerator.
----------------------	---

Methods

[] [Expand table](#)

<code>Dispose()</code>	Releases all resources used by a <code>HashSet<T>.Enumerator</code> object.
------------------------	---

<code>MoveNext()</code>	Advances the enumerator to the next element of the <code>HashSet<T></code> collection.
-------------------------	--

Explicit Interface Implementations

[] [Expand table](#)

<code>IEnumerator.Current</code>	Gets the element at the current position of the enumerator.
----------------------------------	---

<code>IEnumerator.Reset()</code>	Sets the enumerator to its initial position, which is before the first element in the collection.
----------------------------------	---

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the [HashSet<T>](#) collection at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by a [HashSet<T>.Enumerator](#) object.

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [HashSet<T>](#) collection.

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to the [MoveNext](#) method advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator, as an [Object](#).

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IEnumerator.Current](#) does not move the position of the enumerator, and consecutive calls to [IEnumerator.Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling [IEnumator.Reset](#), you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a set of values.

C#

```
public class HashSet<T> : System.Collections.Generic.ICollection<T>,
    System.Collections.Generic.IEnumerable<T>,
    System.Collections.Generic.IReadOnlyCollection<T>,
    System.Collections.Generic.IReadOnlySet<T>, System.Collections.Generic.ISet<T>,
    System.Runtime.Serialization.IDeserializationCallback,
    System.Runtime.Serialization.ISerializable
```

Type Parameters

T

The type of elements in the hash set.

Inheritance [Object](#) → HashSet<T>

Implements [ICollection<T>](#) , [IEnumerable<T>](#) , [IReadOnlyCollection<T>](#) , [ISet<T>](#) ,
[IEnumerable](#) , [IReadOnlySet<T>](#) , [IDeserializationCallback](#) , [ISerializable](#)

Examples

The following example demonstrates how to merge two disparate sets. This example creates two [HashSet<T>](#) objects and populates them with even and odd numbers, respectively. A third [HashSet<T>](#) object is created from the set that contains the even numbers. The example then calls the [UnionWith](#) method, which adds the odd number set to the third set.

C#

```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
```

```

evenNumbers.Add(i * 2);

// Populate oddNumbers with just odd numbers.
oddNumbers.Add((i * 2) + 1);
}

Console.WriteLine("evenNumbers contains {0} elements: ", evenNumbers.Count);
DisplaySet(evenNumbers);

Console.WriteLine("oddNumbers contains {0} elements: ", oddNumbers.Count);
DisplaySet(oddNumbers);

// Create a new HashSet populated with even numbers.
HashSet<int> numbers = new HashSet<int>(evenNumbers);
Console.WriteLine("numbers UnionWith oddNumbers...");
numbers.UnionWith(oddNumbers);

Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

void DisplaySet(HashSet<int> collection)
{
    Console.Write("{");
    foreach (int i in collection)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This example produces output similar to the following:
* evenNumbers contains 5 elements: { 0 2 4 6 8 }
* oddNumbers contains 5 elements: { 1 3 5 7 9 }
* numbers UnionWith oddNumbers...
* numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
*/

```

Remarks

For more information about this API, see [Supplemental API remarks for HashSet<T>](#).

Constructors

[+] Expand table

<code>HashSet<T>()</code>	Initializes a new instance of the <code>HashSet<T></code> class that is empty and uses the default equality comparer for the set type.
---------------------------------	--

<code>HashSet<T>(IEnumerable<T>, IEqualityComparer<T>)</code>	Initializes a new instance of the <code>HashSet<T></code> class that uses the specified equality comparer for the set type, contains elements copied from the specified collection, and has sufficient capacity to accommodate the number of elements copied.
<code>HashSet<T>(IEnumerable<T>)</code>	Initializes a new instance of the <code>HashSet<T></code> class that uses the default equality comparer for the set type, contains elements copied from the specified collection, and has sufficient capacity to accommodate the number of elements copied.
<code>HashSet<T>(IEqualityComparer<T>)</code>	Initializes a new instance of the <code>HashSet<T></code> class that is empty and uses the specified equality comparer for the set type.
<code>HashSet<T>(Int32, IEqualityComparer<T>)</code>	Initializes a new instance of the <code>HashSet<T></code> class that uses the specified equality comparer for the set type, and has sufficient capacity to accommodate <code>capacity</code> elements.
<code>HashSet<T>(Int32)</code>	Initializes a new instance of the <code>HashSet<T></code> class that is empty, but has reserved space for <code>capacity</code> items and uses the default equality comparer for the set type.
<code>HashSet<T>(SerializationInfo, StreamingContext)</code>	Initializes a new instance of the <code>HashSet<T></code> class with serialized data.

Properties

[] [Expand table](#)

<code>Comparer</code>	Gets the <code>IEqualityComparer<T></code> object that is used to determine equality for the values in the set.
<code>Count</code>	Gets the number of elements that are contained in a set.

Methods

[] [Expand table](#)

<code>Add(T)</code>	Adds the specified element to a set.
<code>Clear()</code>	Removes all elements from a <code>HashSet<T></code> object.
<code>Contains(T)</code>	Determines whether a <code>HashSet<T></code> object contains the specified element.
<code>CopyTo(T[], Int32, Int32)</code>	Copies the specified number of elements of a <code>HashSet<T></code> object to an array, starting at the specified array index.

CopyTo(T[], Int32)	Copies the elements of a HashSet<T> object to an array, starting at the specified array index.
CopyTo(T[])	Copies the elements of a HashSet<T> object to an array.
CreateSetComparer()	Returns an IEqualityComparer object that can be used for equality testing of a HashSet<T> object.
EnsureCapacity(Int32)	Ensures that this hash set can hold the specified number of elements without any further expansion of its backing storage.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
ExceptWith(IEnumerable<T>)	Removes all elements in the specified collection from the current HashSet<T> object.
GetEnumerator()	Returns an enumerator that iterates through a HashSet<T> object.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data needed to serialize a HashSet<T> object.
GetType()	Gets the Type of the current instance. (Inherited from Object)
IntersectWith(IEnumerable<T>)	Modifies the current HashSet<T> object to contain only elements that are present in that object and in the specified collection.
IsProperSubsetOf(IEnumerable<T>)	Determines whether a HashSet<T> object is a proper subset of the specified collection.
IsProperSupersetOf(IEnumerable<T>)	Determines whether a HashSet<T> object is a proper superset of the specified collection.
IsSubsetOf(IEnumerable<T>)	Determines whether a HashSet<T> object is a subset of the specified collection.
IsSupersetOf(IEnumerable<T>)	Determines whether a HashSet<T> object is a superset of the specified collection.
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.

Overlaps(IEnumerable<T>)	Determines whether the current HashSet<T> object and a specified collection share common elements.
Remove(T)	Removes the specified element from a HashSet<T> object.
RemoveWhere(Predicate<T>)	Removes all elements that match the conditions defined by the specified predicate from a HashSet<T> collection.
SetEquals(IEnumerable<T>)	Determines whether a HashSet<T> object and the specified collection contain the same elements.
SymmetricExceptWith(IEnumerable<T>)	Modifies the current HashSet<T> object to contain only elements that are present either in that object or in the specified collection, but not both.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity of a HashSet<T> object to the actual number of elements it contains, rounded up to a nearby, implementation-specific value.
TryGetValue(T, T)	Searches the set for a given value and returns the equal value it finds, if any.
UnionWith(IEnumerable<T>)	Modifies the current HashSet<T> object to contain all elements that are present in itself, the specified collection, or both.

Explicit Interface Implementations

[] [Expand table](#)

ICollection<T>.Add(T)	Adds an item to an ICollection<T> object.
ICollection<T>.IsReadOnly	Gets a value indicating whether a collection is read-only.
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through a collection.
IEnumerator<T>.GetEnumerator()	Returns an enumerator that iterates through a collection.

Extension Methods

[] [Expand table](#)

TolmmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
---	---

<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its

	contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.

<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.

<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>)</code>	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.

<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key

<code>Comparer<TKey></code>	selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>,</code>	Correlates the elements of two sequences based on equality of

<code>Func<TOuter, IEnumerable<TInner>, TResult>()</code>	keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.

<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.

<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a

	<p>result selector function on each element therein.</p>
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	<p>Projects each element of a sequence to an IEnumerable<T>, flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	<p>Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T>.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	<p>Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.</p>
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.</p>
Single<TSource>(IEnumerable<TSource>)	<p>Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	<p>Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	<p>Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified</p>

	condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform

	function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.

<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToList<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .

ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an IEnumerable to an IQueryable .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of every node in the source collection.
<code>DescendantNodes<T>(IEnumerable<T>)</code>	Returns a collection of the descendant nodes of every document and element in the source collection.
<code>Descendants<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
<code>Descendants<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the descendant

	elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedSet<T>](#)
- [ISet<T>](#)

HashSet<T> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [HashSet<T>](#) class.

Overloads

[+] [Expand table](#)

HashSet<T>()	Initializes a new instance of the HashSet<T> class that is empty and uses the default equality comparer for the set type.
HashSet<T>(IEnumerable<T>)	Initializes a new instance of the HashSet<T> class that uses the default equality comparer for the set type, contains elements copied from the specified collection, and has sufficient capacity to accommodate the number of elements copied.
HashSet<T>(IEqualityComparer<T>)	Initializes a new instance of the HashSet<T> class that is empty and uses the specified equality comparer for the set type.
HashSet<T>(Int32)	Initializes a new instance of the HashSet<T> class that is empty, but has reserved space for <code>capacity</code> items and uses the default equality comparer for the set type.
HashSet<T>(IEnumerable<T>, IEqualityComparer<T>)	Initializes a new instance of the HashSet<T> class that uses the specified equality comparer for the set type, contains elements copied from the specified collection, and has sufficient capacity to accommodate the number of elements copied.
HashSet<T>(Int32, IEqualityComparer<T>)	Initializes a new instance of the HashSet<T> class that uses the specified equality comparer for the set type, and has sufficient capacity to accommodate <code>capacity</code> elements.
HashSet<T>(SerializationInfo, StreamingContext)	Initializes a new instance of the HashSet<T> class with serialized data.

HashSet<T>()

Initializes a new instance of the [HashSet<T>](#) class that is empty and uses the default equality comparer for the set type.

C#

```
public HashSet();
```

Examples

The following example demonstrates how to create and populate two `HashSet<T>` objects. This example is part of a larger example provided for the [UnionWith](#) method.

C#

```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
    evenNumbers.Add(i * 2);

    // Populate oddNumbers with just odd numbers.
    oddNumbers.Add((i * 2) + 1);
}
```

Remarks

The capacity of a `HashSet<T>` object is the number of elements that the object can hold. A `HashSet<T>` object's capacity automatically increases as elements are added to the object.

This constructor is an O(1) operation.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>(IEnumerable<T>)

Initializes a new instance of the [HashSet<T>](#) class that uses the default equality comparer for the set type, contains elements copied from the specified collection, and has sufficient capacity to accommodate the number of elements copied.

C#

```
public HashSet(System.Collections.Generic.IEnumerable<T> collection);
```

Parameters

collection [IEnumerable<T>](#)

The collection whose elements are copied to the new set.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Examples

The following example shows how to create a [HashSet<T>](#) collection from an existing set. In this example, two sets are created with even and odd integers, respectively. A third [HashSet<T>](#) object is then created from the even integer set.

C#

```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
    evenNumbers.Add(i * 2);

    // Populate oddNumbers with just odd numbers.
    oddNumbers.Add((i * 2) + 1);
}

Console.WriteLine("evenNumbers contains {0} elements: ", evenNumbers.Count);
DisplaySet(evenNumbers);

Console.WriteLine("oddNumbers contains {0} elements: ", oddNumbers.Count);
DisplaySet(oddNumbers);
```

```

// Create a new HashSet populated with even numbers.
HashSet<int> numbers = new HashSet<int>(evenNumbers);
Console.WriteLine("numbers UnionWith oddNumbers...");
numbers.UnionWith(oddNumbers);

Console.Write("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

void DisplaySet(HashSet<int> collection)
{
    Console.Write("{");
    foreach (int i in collection)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This example produces output similar to the following:
* evenNumbers contains 5 elements: { 0 2 4 6 8 }
* oddNumbers contains 5 elements: { 1 3 5 7 9 }
* numbers UnionWith oddNumbers...
* numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
*/

```

Remarks

The capacity of a `HashSet<T>` object is the number of elements that the object can hold. A `HashSet<T>` object's capacity automatically increases as elements are added to the object.

If `collection` contains duplicates, the set will contain one of each unique element. No exception will be thrown. Therefore, the size of the resulting set is not identical to the size of `collection`.

This constructor is an $O(n)$ operation, where n is the number of elements in the `collection` parameter.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>(IEqualityComparer<T>)

Initializes a new instance of the [HashSet<T>](#) class that is empty and uses the specified equality comparer for the set type.

C#

```
public HashSet(System.Collections.Generic.IEqualityComparer<T> comparer);
```

Parameters

comparer [IEqualityComparer<T>](#)

The [IEqualityComparer<T>](#) implementation to use when comparing values in the set, or `null` to use the default [EqualityComparer<T>](#) implementation for the set type.

Remarks

The capacity of a [HashSet<T>](#) object is the number of elements that the object can hold. A [HashSet<T>](#) object's capacity automatically increases as elements are added to the object.

This constructor is an O(1) operation.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>(Int32)

Initializes a new instance of the [HashSet<T>](#) class that is empty, but has reserved space for `capacity` items and uses the default equality comparer for the set type.

C#

```
public HashSet(int capacity);
```

Parameters

capacity `Int32`

The initial size of the [HashSet<T>](#).

Remarks

Since resizes are relatively expensive (require rehashing), this attempts to minimize the need to resize by setting the initial capacity based on the value of the `capacity`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.7.2, 4.8, 4.8.1
.NET Standard	2.1

HashSet<T>(IEnumerable<T>, IEqualityComparer<T>)

Initializes a new instance of the [HashSet<T>](#) class that uses the specified equality comparer for the set type, contains elements copied from the specified collection, and has sufficient capacity to accommodate the number of elements copied.

C#

```
public HashSet(System.Collections.Generic.IEnumerable<T> collection,
System.Collections.Generic.IEqualityComparer<T>? comparer);
```

Parameters

collection [IEnumerable<T>](#)

The collection whose elements are copied to the new set.

comparer [IEqualityComparer<T>](#)

The [IEqualityComparer<T>](#) implementation to use when comparing values in the set, or `null` to use the default [EqualityComparer<T>](#) implementation for the set type.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Examples

The following example uses a supplied [IEqualityComparer<T>](#) to allow case-insensitive comparisons on the elements of a [HashSet<T>](#) collection of vehicle types.

C#

```
HashSet<string> allVehicles = new HashSet<string>
(StringComparer.OrdinalIgnoreCase);
List<string> someVehicles = new List<string>();

someVehicles.Add("Planes");
someVehicles.Add("Trains");
someVehicles.Add("Automobiles");

// Add in the vehicles contained in the someVehicles list.
allVehicles.UnionWith(someVehicles);

Console.WriteLine("The current HashSet contains:\n");
foreach (string vehicle in allVehicles)
{
    Console.WriteLine(vehicle);
}

allVehicles.Add("Ships");
allVehicles.Add("Motorcycles");
allVehicles.Add("Rockets");
allVehicles.Add("Helicopters");
allVehicles.Add("Submarines");
```

```
Console.WriteLine("\nThe updated HashSet contains:\n");
foreach (string vehicle in allVehicles)
{
    Console.WriteLine(vehicle);
}

// Verify that the 'All Vehicles' set contains at least the vehicles in
// the 'Some Vehicles' list.
if (allVehicles.IsSupersetOf(someVehicles))
{
    Console.Write("\nThe 'All' vehicles set contains everything in ");
    Console.WriteLine("'Some' vechicles list.");
}

// Check for Rockets. Here the OrdinalIgnoreCase comparer will compare
// true for the mixed-case vehicle type.
if (allVehicles.Contains("roCKeTs"))
{
    Console.WriteLine("\nThe 'All' vehicles set contains 'roCKeTs'");
}

allVehicles.ExceptWith(someVehicles);
Console.WriteLine("\nThe excepted HashSet contains:\n");
foreach (string vehicle in allVehicles)
{
    Console.WriteLine(vehicle);
}

// Remove all the vehicles that are not 'super cool'.
allVehicles.RemoveWhere(isNotSuperCool);

Console.WriteLine("\nThe super cool vehicles are:\n");
foreach (string vehicle in allVehicles)
{
    Console.WriteLine(vehicle);
}

// Predicate to determine vehicle 'coolness'.
bool isNotSuperCool(string vehicle)
{
    bool superCool = (vehicle == "Helicopters") || (vehicle == "Motorcycles");

    return !superCool;
}

// The program writes the following output to the console.
//
// The current HashSet contains:
//
// Planes
// Trains
// Automobiles
//
// The updated HashSet contains:
```

```

// 
// Planes
// Trains
// Automobiles
// Ships
// Motorcycles
// Rockets
// Helicopters
// Submarines
//
// The 'All' vehicles set contains everything in 'Some' vechicles list.
//
// The 'All' vehicles set contains 'roCKeTs'
//
// The excepted HashSet contains:
//
// Ships
// Motorcycles
// Rockets
// Helicopters
// Submarines
//
// The super cool vehicles are:
//
// Motorcycles
// Helicopters

```

Remarks

The capacity of a `HashSet<T>` object is the number of elements that the object can hold. A `HashSet<T>` object's capacity automatically increases as elements are added to the object.

If `collection` contains duplicates, the set will contain one of each unique element. No exception will be thrown. Therefore, the size of the resulting set is not identical to the size of `collection`.

This constructor is an $O(n)$ operation, where `n` is the number of elements in the `collection` parameter.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>(Int32, IEqualityComparer<T>)

Initializes a new instance of the [HashSet<T>](#) class that uses the specified equality comparer for the set type, and has sufficient capacity to accommodate `capacity` elements.

C#

```
public HashSet(int capacity, System.Collections.Generic.IEqualityComparer<T> comparer);
```

Parameters

capacity `Int32`

The initial size of the [HashSet<T>](#).

comparer `IEqualityComparer<T>`

The [IEqualityComparer<T>](#) implementation to use when comparing values in the set, or null (Nothing in Visual Basic) to use the default [IEqualityComparer<T>](#) implementation for the set type.

Remarks

Since resizes are relatively expensive (require rehashing), this attempts to minimize the need to resize by setting the initial capacity based on the value of the `capacity`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.7.2, 4.8, 4.8.1
.NET Standard	2.1

HashSet<T>(SerializationInfo, StreamingContext)

Initializes a new instance of the [HashSet<T>](#) class with serialized data.

C#

```
protected HashSet(System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information required to serialize the [HashSet<T>](#) object.

context [StreamingContext](#)

A [StreamingContext](#) structure that contains the source and destination of the serialized stream associated with the [HashSet<T>](#) object.

Remarks

This constructor is called during deserialization to reconstitute an object that is transmitted over a stream. For more information, see [XML and SOAP Serialization](#).

Applies to

▼ .NET 10 and other versions

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

HashSet<T>.Comparer Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the [IEqualityComparer<T>](#) object that is used to determine equality for the values in the set.

C#

```
public System.Collections.Generic.IEqualityComparer<T> Comparer { get; }
```

Property Value

[IEqualityComparer<T>](#)

The [IEqualityComparer<T>](#) object that is used to determine equality for the values in the set.

Remarks

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements that are contained in a set.

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements that are contained in the set.

Implements

[Count](#), [Count](#)

Examples

The following example demonstrates how to create, populate, and manipulate two [HashSet<T>](#) objects. In this example, both the contents of the set and [Count](#) display to the console.

C#

```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
    evenNumbers.Add(i * 2);

    // Populate oddNumbers with just odd numbers.
    oddNumbers.Add((i * 2) + 1);
}

Console.WriteLine("evenNumbers contains {0} elements: ", evenNumbers.Count);
DisplaySet(evenNumbers);
```

```

Console.WriteLine("oddNumbers contains {0} elements: ", oddNumbers.Count);
DisplaySet(oddNumbers);

// Create a new HashSet populated with even numbers.
HashSet<int> numbers = new HashSet<int>(evenNumbers);
Console.WriteLine("numbers UnionWith oddNumbers...");
numbers.UnionWith(oddNumbers);

Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

void DisplaySet(HashSet<int> collection)
{
    Console.Write("{");
    foreach (int i in collection)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");

/*
 * This example produces output similar to the following:
 * evenNumbers contains 5 elements: { 0 2 4 6 8 }
 * oddNumbers contains 5 elements: { 1 3 5 7 9 }
 * numbers UnionWith oddNumbers...
 * numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
 */
}

```

Remarks

The capacity of a [HashSet<T>](#) object is the number of elements that the object can hold. A [HashSet<T>](#) object's capacity automatically increases as elements are added to the object.

The capacity is always greater than or equal to [Count](#). If [Count](#) exceeds the capacity while adding elements, the capacity is set to the first prime number that is greater than double the previous capacity.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds the specified element to a set.

C#

```
public bool Add(T item);
```

Parameters

item T

The element to add to the set.

Returns

Boolean

`true` if the element is added to the [HashSet<T>](#) object; `false` if the element is already present.

Implements

[Add\(T\)](#)

Examples

The following example demonstrates how to create and populate two [HashSet<T>](#) objects. This example is part of a larger example provided for the [UnionWith](#) method.

C#

```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
    evenNumbers.Add(i * 2);
```

```
// Populate oddNumbers with just odd numbers.  
oddNumbers.Add((i * 2) + 1);  
}
```

Remarks

If [Count](#) already equals the capacity of the [HashSet<T>](#) object, the capacity is automatically adjusted to accommodate the new item.

If [Count](#) is less than the capacity of the internal array, this method is an O(1) operation. If the [HashSet<T>](#) object must be resized, this method becomes an O([n](#)) operation, where [n](#) is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements from a [HashSet<T>](#) object.

C#

```
public void Clear();
```

Implements

[Clear\(\)](#)

Examples

The following example creates and populates a [HashSet<T>](#) collection, then clears it and releases the memory referenced by the collection.

C#

```
HashSet<int> Numbers = new HashSet<int>();

for (int i = 0; i < 10; i++)
{
    Numbers.Add(i);
}

Console.WriteLine("Numbers contains {0} elements: ", Numbers.Count);
DisplaySet(Numbers);

Numbers.Clear();
Numbers.TrimExcess();

Console.WriteLine("Numbers contains {0} elements: ", Numbers.Count);
DisplaySet(Numbers);

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
}
```

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

/* This example produces output similar to the following:
* Numbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
* Numbers contains 0 elements: { }
*/
```

Remarks

[Count](#) is set to zero and references to other objects from elements of the collection are also released. The capacity remains unchanged until a call to [TrimExcess](#) is made.

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [HashSet<T>](#) object contains the specified element.

C#

```
public bool Contains(T item);
```

Parameters

item T

The element to locate in the [HashSet<T>](#) object.

Returns

Boolean

`true` if the [HashSet<T>](#) object contains the specified element; otherwise, `false`.

Implements

[Contains\(T\)](#) , [Contains\(T\)](#)

Examples

The following example demonstrates how to remove values from a [HashSet<T>](#) collection using the [Remove](#) method. In this example, the [Contains](#) method verifies that the set contains a value before removing it.

C#

```
HashSet<int> numbers = new HashSet<int>();

for (int i = 0; i < 20; i++) {
    numbers.Add(i);
}

// Display all the numbers in the hash table.
```

```

Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

// Remove all odd numbers.
numbers.RemoveWhere(IsOdd);
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

// Check if the hash table contains 0 and, if so, remove it.
if (numbers.Contains(0)) {
    numbers.Remove(0);
}
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

bool IsOdd(int i)
{
    return ((i % 2) == 1);
}

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
        Console.Write(" {0}", i);

    Console.WriteLine(" }");
}

// This example displays the following output:
//     numbers contains 20 elements: { 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 }
//     numbers contains 10 elements: { 0 2 4 6 8 10 12 14 16 18 }
//     numbers contains 9 elements: { 2 4 6 8 10 12 14 16 18 }

```

Remarks

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of a [HashSet<T>](#) collection to an array.

Overloads

[+] [Expand table](#)

CopyTo(T[])	Copies the elements of a HashSet<T> object to an array.
CopyTo(T[], Int32)	Copies the elements of a HashSet<T> object to an array, starting at the specified array index.
CopyTo(T[], Int32, Int32)	Copies the specified number of elements of a HashSet<T> object to an array, starting at the specified array index.

CopyTo(T[])

Copies the elements of a [HashSet<T>](#) object to an array.

C#

```
public void CopyTo(T[] array);
```

Parameters

array [T\[\]](#)

The one-dimensional array that is the destination of the elements copied from the [HashSet<T>](#) object. The array must have zero-based indexing.

Exceptions

[ArgumentNullException](#)

`array` is `null`.

Remarks

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

CopyTo(T[], Int32)

Copies the elements of a [HashSet<T>](#) object to an array, starting at the specified array index.

C#

```
public void CopyTo(T[] array, int arrayIndex);
```

Parameters

array T[]

The one-dimensional array that is the destination of the elements copied from the [HashSet<T>](#) object. The array must have zero-based indexing.

arrayIndex Int32

The zero-based index in **array** at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

ArgumentNullException

`array` is `null`.

ArgumentOutOfRangeException

`arrayIndex` is less than 0.

ArgumentException

`arrayIndex` is greater than the length of the destination `array`.

Remarks

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

CopyTo(T[], Int32, Int32)

Copies the specified number of elements of a [HashSet<T>](#) object to an array, starting at the specified array index.

C#

```
public void CopyTo(T[] array, int arrayIndex, int count);
```

Parameters

`array` `T[]`

The one-dimensional array that is the destination of the elements copied from the [HashSet<T>](#) object. The array must have zero-based indexing.

arrayIndex [Int32](#)

The zero-based index in `array` at which copying begins.

count [Int32](#)

The number of elements to copy to `array`.

Exceptions

[ArgumentNullException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#)

`arrayIndex` is less than 0.

-or-

`count` is less than 0.

[ArgumentException](#)

`arrayIndex` is greater than the length of the destination `array`.

-or-

`count` is greater than the available space from `arrayIndex` to the end of the destination `array`.

Remarks

This method is an $O(n)$ operation, where `n` is `count`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.CreateSetComparer Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an [IEqualityComparer](#) object that can be used for equality testing of a [HashSet<T>](#) object.

C#

```
public static  
System.Collections.Generic.IEqualityComparer<System.Collections.Generic.HashSet<T>  
> CreateSetComparer();
```

Returns

[IEqualityComparer<HashSet<T>>](#)

An [IEqualityComparer](#) object that can be used for deep equality testing of the [HashSet<T>](#) object.

Remarks

The [IEqualityComparer](#) object checks for equality at only one level; however, you can chain together comparers at additional levels to perform deeper equality testing.

Calling this method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

HashSet<T>.EnsureCapacity(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Ensures that this hash set can hold the specified number of elements without any further expansion of its backing storage.

C#

```
public int EnsureCapacity(int capacity);
```

Parameters

capacity [Int32](#)

The minimum capacity to ensure.

Returns

[Int32](#)

The new capacity of this instance.

Exceptions

[ArgumentOutOfRangeException](#)

capacity is less than zero.

Applies to

Product	Versions
.NET	Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

HashSet<T>.ExceptWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements in the specified collection from the current [HashSet<T>](#) object.

C#

```
public void ExceptWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection of items to remove from the [HashSet<T>](#) object.

Implements

[ExceptWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example creates two [HashSet<T>](#) collections with overlapping sets of data. The lower range of values is then removed from the larger set using the [ExceptWith](#) method.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> highNumbers = new HashSet<int>();

for (int i = 0; i < 6; i++)
{
    lowNumbers.Add(i);
}
```

```

for (int i = 3; i < 10; i++)
{
    highNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("highNumbers contains {0} elements: ", highNumbers.Count);
DisplaySet(highNumbers);

Console.WriteLine("highNumbers ExceptWith lowNumbers...");
highNumbers.ExceptWith(lowNumbers);

Console.WriteLine("highNumbers contains {0} elements: ", highNumbers.Count);
DisplaySet(highNumbers);

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This example provides output similar to the following:
 * lowNumbers contains 6 elements: { 0 1 2 3 4 5 }
 * highNumbers contains 7 elements: { 3 4 5 6 7 8 9 }
 * highNumbers ExceptWith lowNumbers...
 * highNumbers contains 4 elements: { 6 7 8 9 }
 */

```

Remarks

The [ExceptWith](#) method is the equivalent of mathematical set subtraction.

This method is an $O(n)$ operation, where n is the number of elements in the `other` parameter.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a [HashSet<T>](#) object.

C#

```
public System.Collections.Generic.HashSet<T>.Enumerator GetEnumerator();
```

Returns

[HashSet<T>.Enumerator](#)

A [HashSet<T>.Enumerator](#) object for the [HashSet<T>](#) object.

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator object instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.GetObjectData(SerializationInfo, StreamingContext) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and returns the data needed to serialize a [HashSet<T>](#) object.

C#

```
public virtual void GetObjectData(System.Runtime.Serialization.SerializationInfo  
info, System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information required to serialize the [HashSet<T>](#) object.

context [StreamingContext](#)

A [StreamingContext](#) structure that contains the source and destination of the serialized stream associated with the [HashSet<T>](#) object.

Implements

[GetObjectData\(SerializationInfo, StreamingContext\)](#)

Exceptions

[ArgumentNullException](#)

`info` is `null`.

Remarks

Calling this method is an O(`n`) operation, where `n` is [Count](#).

Applies to

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

HashSet<T>.IntersectWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Modifies the current [HashSet<T>](#) object to contain only elements that are present in that object and in the specified collection.

C#

```
public void IntersectWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Implements

[IntersectWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

If the collection represented by the `other` parameter is a [HashSet<T>](#) collection with the same equality comparer as the current [HashSet<T>](#) object, this method is an $O(n)$ operation.

Otherwise, this method is an $O(n + m)$ operation, where `n` is [Count](#) and `m` is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.IsProperSubsetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [HashSet<T>](#) object is a proper subset of the specified collection.

C#

```
public bool IsProperSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [HashSet<T>](#) object is a proper subset of `other`; otherwise, `false`.

Implements

[IsProperSubsetOf\(IEnumerable<T>\)](#) , [IsProperSubsetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example creates two disparate [HashSet<T>](#) objects and compares them to each other. In this example, `lowNumbers` is both a subset and a proper subset of `allNumbers` until `allNumbers` is modified, using the [IntersectWith](#) method, to contain only values that are

present in both sets. Once `allNumbers` and `lowNumbers` are identical, `lowNumbers` is still a subset of `allNumbers` but is no longer a proper subset.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> allNumbers = new HashSet<int>();

for (int i = 1; i < 5; i++)
{
    lowNumbers.Add(i);
}

for (int i = 0; i < 10; i++)
{
    allNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("lowNumbers overlaps allNumbers: {0}",
    lowNumbers.Overlaps(allNumbers));

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

// Modify allNumbers to remove numbers that are not in lowNumbers.
allNumbers.IntersectWith(lowNumbers);
Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing with the modified set.
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
```

```

Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");

/*
 * This code example produces output similar to the following:
 * lowNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
 * lowNumbers overlaps allNumbers: True
 * allNumbers and lowNumbers are equal sets: False
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: True
 * allNumbers is a proper superset of lowNumbers: True
 * allNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers and lowNumbers are equal sets: True
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: False
 * allNumbers is a proper superset of lowNumbers: False
 */
}

```

Remarks

An empty set is a proper subset of any other collection. Therefore, this method returns `true` if the collection represented by the current `HashSet<T>` object is empty unless the `other` parameter is also an empty set.

This method always returns `false` if `Count` is greater than or equal to the number of elements in `other`.

If the collection represented by `other` is a `HashSet<T>` collection with the same equality comparer as the current `HashSet<T>` object, then this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is `Count` and `m` is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.IsProperSupersetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [HashSet<T>](#) object is a proper superset of the specified collection.

C#

```
public bool IsProperSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [HashSet<T>](#) object is a proper superset of `other`; otherwise, `false`.

Implements

[IsProperSupersetOf\(IEnumerable<T>\)](#), [IsProperSupersetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example creates two disparate [HashSet<T>](#) objects and compares them to each other. In this example, `allNumbers` is both a superset and a proper superset of `lowNumbers` until `allNumbers` is modified, using the [IntersectWith](#) method, to contain only values that are

present in both sets. Once `allNumbers` and `lowNumbers` are identical, `allNumbers` is still a superset of `lowNumbers` but is no longer a proper superset.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> allNumbers = new HashSet<int>();

for (int i = 1; i < 5; i++)
{
    lowNumbers.Add(i);
}

for (int i = 0; i < 10; i++)
{
    allNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("lowNumbers overlaps allNumbers: {0}",
    lowNumbers.Overlaps(allNumbers));

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

// Modify allNumbers to remove numbers that are not in lowNumbers.
allNumbers.IntersectWith(lowNumbers);
Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing with the modified set.
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
```

```

Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");

/*
 * This code example produces output similar to the following:
 * lowNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
 * lowNumbers overlaps allNumbers: True
 * allNumbers and lowNumbers are equal sets: False
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: True
 * allNumbers is a proper superset of lowNumbers: True
 * allNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers and lowNumbers are equal sets: True
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: False
 * allNumbers is a proper superset of lowNumbers: False
 */

```

Remarks

An empty set is a proper superset of any other collection. Therefore, this method returns `true` if the collection represented by the `other` parameter is empty unless the current `HashSet<T>` collection is also empty.

This method always returns `false` if `Count` is less than or equal to the number of elements in `other`.

If the collection represented by `other` is a `HashSet<T>` collection with the same equality comparer as the current `HashSet<T>` object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is the number of elements in `other` and `m` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.IsSubsetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [HashSet<T>](#) object is a subset of the specified collection.

C#

```
public bool IsSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [HashSet<T>](#) object is a subset of `other`; otherwise, `false`.

Implements

[IsSubsetOf\(IEnumerable<T>\)](#), [IsSubsetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example creates two disparate [HashSet<T>](#) objects and compares them to each other. In this example, `lowNumbers` is both a subset and a proper subset of `allNumbers` until `allNumbers` is modified, using the [IntersectWith](#) method, to contain only values that are

present in both sets. Once `allNumbers` and `lowNumbers` are identical, `lowNumbers` is still a subset of `allNumbers` but is no longer a proper subset.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> allNumbers = new HashSet<int>();

for (int i = 1; i < 5; i++)
{
    lowNumbers.Add(i);
}

for (int i = 0; i < 10; i++)
{
    allNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("lowNumbers overlaps allNumbers: {0}",
    lowNumbers.Overlaps(allNumbers));

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

// Modify allNumbers to remove numbers that are not in lowNumbers.
allNumbers.IntersectWith(lowNumbers);
Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing with the modified set.
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
```

```

Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");

/*
 * This code example produces output similar to the following:
 * lowNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
 * lowNumbers overlaps allNumbers: True
 * allNumbers and lowNumbers are equal sets: False
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: True
 * allNumbers is a proper superset of lowNumbers: True
 * allNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers and lowNumbers are equal sets: True
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: False
 * allNumbers is a proper superset of lowNumbers: False
 */
}

```

Remarks

An empty set is a subset of any other collection, including an empty set; therefore, this method returns `true` if the collection represented by the current `HashSet<T>` object is empty, even if the `other` parameter is an empty set.

This method always returns `false` if `Count` is greater than the number of elements in `other`.

If the collection represented by `other` is a `HashSet<T>` collection with the same equality comparer as the current `HashSet<T>` object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is `Count` and `m` is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.IsSupersetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [HashSet<T>](#) object is a superset of the specified collection.

C#

```
public bool IsSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [HashSet<T>](#) object is a superset of `other`; otherwise, `false`.

Implements

[IsSupersetOf\(IEnumerable<T>\)](#), [IsSupersetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example creates two disparate [HashSet<T>](#) objects and compares them to each other. In this example, `allNumbers` is both a superset and a proper superset of `lowNumbers` until `allNumbers` is modified, using the [IntersectWith](#) method, to contain only values that are

present in both sets. Once `allNumbers` and `lowNumbers` are identical, `allNumbers` is still a superset of `lowNumbers` but is no longer a proper superset.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> allNumbers = new HashSet<int>();

for (int i = 1; i < 5; i++)
{
    lowNumbers.Add(i);
}

for (int i = 0; i < 10; i++)
{
    allNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("lowNumbers overlaps allNumbers: {0}",
    lowNumbers.Overlaps(allNumbers));

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

// Modify allNumbers to remove numbers that are not in lowNumbers.
allNumbers.IntersectWith(lowNumbers);
Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing with the modified set.
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
```

```

Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");

/*
 * This code example produces output similar to the following:
 * lowNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
 * lowNumbers overlaps allNumbers: True
 * allNumbers and lowNumbers are equal sets: False
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: True
 * allNumbers is a proper superset of lowNumbers: True
 * allNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers and lowNumbers are equal sets: True
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: False
 * allNumbers is a proper superset of lowNumbers: False
 */
}

```

Remarks

All collections, including the empty set, are supersets of the empty set. Therefore, this method returns `true` if the collection represented by the `other` parameter is empty, even if the current `HashSet<T>` object is empty.

This method always returns `false` if `Count` is less than the number of elements in `other`.

If the collection represented by `other` is a `HashSet<T>` collection with the same equality comparer as the current `HashSet<T>` object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is the number of elements in `other` and `m` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.OnDeserialization(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and raises the deserialization event when the deserialization is complete.

C#

```
public virtual void OnDeserialization(object? sender);
```

Parameters

sender [Object](#)

The source of the deserialization event.

Implements

[OnDeserialization\(Object\)](#)

Exceptions

[SerializationException](#)

The [SerializationInfo](#) object associated with the current [HashSet<T>](#) object is invalid.

Remarks

Calling this method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	2.0, 2.1

HashSet<T>.Overlaps(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the current [HashSet<T>](#) object and a specified collection share common elements.

C#

```
public bool Overlaps(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [HashSet<T>](#) object and `other` share at least one common element; otherwise, `false`.

Implements

[Overlaps\(IEnumerable<T>\)](#) , [Overlaps\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example creates two disparate [HashSet<T>](#) objects and compares them to each other. In this example, `allNumbers` and `lowNumbers` are shown to share common elements

using the [Overlaps](#) method.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> allNumbers = new HashSet<int>();

for (int i = 1; i < 5; i++)
{
    lowNumbers.Add(i);
}

for (int i = 0; i < 10; i++)
{
    allNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("lowNumbers overlaps allNumbers: {0}",
    lowNumbers.Overlaps(allNumbers));

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

// Modify allNumbers to remove numbers that are not in lowNumbers.
allNumbers.IntersectWith(lowNumbers);
Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing with the modified set.
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
```

```

Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");

/*
 * This code example produces output similar to the following:
 * lowNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
 * lowNumbers overlaps allNumbers: True
 * allNumbers and lowNumbers are equal sets: False
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: True
 * allNumbers is a proper superset of lowNumbers: True
 * allNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers and lowNumbers are equal sets: True
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: False
 * allNumbers is a proper superset of lowNumbers: False
*/
}

```

Remarks

This method is an $O(n)$ operation, where n is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.Remove(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the specified element from a [HashSet<T>](#) object.

C#

```
public bool Remove(T item);
```

Parameters

item T

The element to remove.

Returns

Boolean

`true` if the element is successfully found and removed; otherwise, `false`. This method returns `false` if `item` is not found in the [HashSet<T>](#) object.

Implements

[Remove\(T\)](#)

Examples

The following example demonstrates how to remove values from a [HashSet<T>](#) collection using the [Remove](#) method. In this example, zero is arbitrarily removed from the [HashSet<T>](#) collection.

C#

```
HashSet<int> numbers = new HashSet<int>();  
  
for (int i = 0; i < 20; i++) {  
    numbers.Add(i);  
}  
numbers.Remove(0);
```

```

// Display all the numbers in the hash table.
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

// Remove all odd numbers.
numbers.RemoveWhere(IsOdd);
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

// Check if the hash table contains 0 and, if so, remove it.
if (numbers.Contains(0)) {
    numbers.Remove(0);
}
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

bool IsOdd(int i)
{
    return ((i % 2) == 1);
}

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
        Console.Write(" {0}", i);

    Console.WriteLine(" }");
}

// This example displays the following output:
//     numbers contains 20 elements: { 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 }
//     numbers contains 10 elements: { 0 2 4 6 8 10 12 14 16 18 }
//     numbers contains 9 elements: { 2 4 6 8 10 12 14 16 18 }

```

Remarks

If the `HashSet<T>` object does not contain the specified element, the object remains unchanged. No exception is thrown.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.RemoveWhere(Predicate<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements that match the conditions defined by the specified predicate from a [HashSet<T>](#) collection.

C#

```
public int RemoveWhere(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the elements to remove.

Returns

[Int32](#)

The number of elements that were removed from the [HashSet<T>](#) collection.

Exceptions

[ArgumentNullException](#)

`match` is `null`.

Examples

The following example demonstrates how to remove values from a [HashSet<T>](#) collection using the [Remove](#) method. In this example, all odd integers are removed from the [HashSet<T>](#) collection as specified by the `match` delegate.

C#

```
HashSet<int> numbers = new HashSet<int>();
```

```

for (int i = 0; i < 20; i++) {
    numbers.Add(i);
}

// Display all the numbers in the hash table.
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

// Remove all odd numbers.
numbers.RemoveWhere(IsOdd);
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

// Check if the hash table contains 0 and, if so, remove it.
if (numbers.Contains(0)) {
    numbers.Remove(0);
}
Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

bool IsOdd(int i)
{
    return ((i % 2) == 1);
}

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
        Console.Write(" {0}", i);

    Console.WriteLine(" }");
}

// This example displays the following output:
//     numbers contains 20 elements: { 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 }
//     numbers contains 10 elements: { 0 2 4 6 8 10 12 14 16 18 }
//     numbers contains 9 elements: { 2 4 6 8 10 12 14 16 18 }

```

Remarks

Calling this method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.SetEquals(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [HashSet<T>](#) object and the specified collection contain the same elements.

C#

```
public bool SetEquals(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [HashSet<T>](#) object is equal to `other`; otherwise, `false`.

Implements

[SetEquals\(IEnumerable<T>\)](#) , [SetEquals\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example creates two disparate [HashSet<T>](#) objects and compares them to each other. Initially, the two sets are not equal, which is demonstrated by using the [SetEquals](#) method. The `allNumbers` [HashSet<T>](#) object is then modified, after which the sets are equal.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> allNumbers = new HashSet<int>();

for (int i = 1; i < 5; i++)
{
    lowNumbers.Add(i);
}

for (int i = 0; i < 10; i++)
{
    allNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("lowNumbers overlaps allNumbers: {0}",
    lowNumbers.Overlaps(allNumbers));

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));

// Modify allNumbers to remove numbers that are not in lowNumbers.
allNumbers.IntersectWith(lowNumbers);
Console.WriteLine("allNumbers contains {0} elements: ", allNumbers.Count);
DisplaySet(allNumbers);

Console.WriteLine("allNumbers and lowNumbers are equal sets: {0}",
    allNumbers.SetEquals(lowNumbers));

// Show the results of sub/superset testing with the modified set.
Console.WriteLine("lowNumbers is a subset of allNumbers: {0}",
    lowNumbers.IsSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a superset of lowNumbers: {0}",
    allNumbers.IsSupersetOf(lowNumbers));
Console.WriteLine("lowNumbers is a proper subset of allNumbers: {0}",
    lowNumbers.IsProperSubsetOf(allNumbers));
Console.WriteLine("allNumbers is a proper superset of lowNumbers: {0}",
    allNumbers.IsProperSupersetOf(lowNumbers));
```

```

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This code example produces output similar to the following:
 * lowNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
 * lowNumbers overlaps allNumbers: True
 * allNumbers and lowNumbers are equal sets: False
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: True
 * allNumbers is a proper superset of lowNumbers: True
 * allNumbers contains 4 elements: { 1 2 3 4 }
 * allNumbers and lowNumbers are equal sets: True
 * lowNumbers is a subset of allNumbers: True
 * allNumbers is a superset of lowNumbers: True
 * lowNumbers is a proper subset of allNumbers: False
 * allNumbers is a proper superset of lowNumbers: False
*/

```

Remarks

The [SetEquals](#) method ignores duplicate entries and the order of elements in the `other` parameter.

If the collection represented by `other` is a [HashSet<T>](#) collection with the same equality comparer as the current [HashSet<T>](#) object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is the number of elements in `other` and `m` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

HashSet<T>.SymmetricExceptWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Modifies the current [HashSet<T>](#) object to contain only elements that are present either in that object or in the specified collection, but not both.

C#

```
public void SymmetricExceptWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Implements

[SymmetricExceptWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

other is `null`.

Examples

The following example creates two [HashSet<T>](#) collections with overlapping sets of data. The set that contains the lower values is then modified, using the [SymmetricExceptWith](#) method, to contain only the values that are not present in both sets.

C#

```
HashSet<int> lowNumbers = new HashSet<int>();
HashSet<int> highNumbers = new HashSet<int>();

for (int i = 0; i < 6; i++)
```

```

{
    lowNumbers.Add(i);
}

for (int i = 3; i < 10; i++)
{
    highNumbers.Add(i);
}

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

Console.WriteLine("highNumbers contains {0} elements: ", highNumbers.Count);
DisplaySet(highNumbers);

Console.WriteLine("lowNumbers SymmetricExceptWith highNumbers...");
lowNumbers.SymmetricExceptWith(highNumbers);

Console.WriteLine("lowNumbers contains {0} elements: ", lowNumbers.Count);
DisplaySet(lowNumbers);

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This example provides output similar to the following:
 * lowNumbers contains 6 elements: { 0 1 2 3 4 5 }
 * highNumbers contains 7 elements: { 3 4 5 6 7 8 9 }
 * lowNumbers SymmetricExceptWith highNumbers...
 * lowNumbers contains 7 elements: { 0 1 2 8 7 6 9 }
 */

```

Remarks

If the `other` parameter is a `HashSet<T>` collection with the same equality comparer as the current `HashSet<T>` object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is the number of elements in `other` and `m` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.TrimExcess Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Overloads

 [Expand table](#)

[TrimExcess\(\)](#) Sets the capacity of a [HashSet<T>](#) object to the actual number of elements it contains, rounded up to a nearby, implementation-specific value.

TrimExcess()

Sets the capacity of a [HashSet<T>](#) object to the actual number of elements it contains, rounded up to a nearby, implementation-specific value.

C#

```
public void TrimExcess();
```

Examples

The following example creates and populates a [HashSet<T>](#) collection, and then clears the collection and releases the memory referenced by it.

C#

```
HashSet<int> Numbers = new HashSet<int>();

for (int i = 0; i < 10; i++)
{
    Numbers.Add(i);
}

Console.WriteLine("Numbers contains {0} elements: ", Numbers.Count);
DisplaySet(Numbers);

Numbers.Clear();
Numbers.TrimExcess();
```

```

Console.WriteLine("Numbers contains {0} elements: ", Numbers.Count);
DisplaySet(Numbers);

void DisplaySet(HashSet<int> set)
{
    Console.Write("{");
    foreach (int i in set)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This example produces output similar to the following:
* Numbers contains 10 elements: { 0 1 2 3 4 5 6 7 8 9 }
* Numbers contains 0 elements: { }
*/

```

Remarks

You can use the [TrimExcess](#) method to minimize a `HashSet<T>` object's memory overhead once it is known that no new elements will be added. To completely clear a `HashSet<T>` object and release all memory referenced by it, call this method after calling the [Clear](#) method.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.TryGetValue(T, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches the set for a given value and returns the equal value it finds, if any.

C#

```
public bool TryGetValue(T equalValue, out T actualValue);
```

Parameters

equalValue **T**

The value to search for.

actualValue **T**

The value from the set that the search found, or the default value of T when the search yielded no match.

Returns

[Boolean](#)

A value indicating whether the search was successful.

Remarks

This can be useful when you want to reuse a previously stored reference instead of a newly constructed one (so that more sharing of references can occur) or to look up a value that has more complete data than the value you currently have, although their comparer functions indicate they are equal.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.7.2, 4.8, 4.8.1
.NET Standard	2.1

HashSet<T>.UnionWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Modifies the current [HashSet<T>](#) object to contain all elements that are present in itself, the specified collection, or both.

C#

```
public void UnionWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [HashSet<T>](#) object.

Implements

[UnionWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

other is `null`.

Examples

The following example demonstrates how to merge two disparate sets. This example creates two [HashSet<T>](#) objects, and populates them with even and odd numbers, respectively. A third [HashSet<T>](#) object is created from the set that contains the even numbers. The example then calls the [UnionWith](#) method, which adds the odd number set to the third set.

C#

```
HashSet<int> evenNumbers = new HashSet<int>();
HashSet<int> oddNumbers = new HashSet<int>();
```

```

for (int i = 0; i < 5; i++)
{
    // Populate numbers with just even numbers.
    evenNumbers.Add(i * 2);

    // Populate oddNumbers with just odd numbers.
    oddNumbers.Add((i * 2) + 1);
}

Console.WriteLine("evenNumbers contains {0} elements: ", evenNumbers.Count);
DisplaySet(evenNumbers);

Console.WriteLine("oddNumbers contains {0} elements: ", oddNumbers.Count);
DisplaySet(oddNumbers);

// Create a new HashSet populated with even numbers.
HashSet<int> numbers = new HashSet<int>(evenNumbers);
Console.WriteLine("numbers UnionWith oddNumbers...");
numbers.UnionWith(oddNumbers);

Console.WriteLine("numbers contains {0} elements: ", numbers.Count);
DisplaySet(numbers);

void DisplaySet(HashSet<int> collection)
{
    Console.Write("{");
    foreach (int i in collection)
    {
        Console.Write(" {0}", i);
    }
    Console.WriteLine(" }");
}

/* This example produces output similar to the following:
* evenNumbers contains 5 elements: { 0 2 4 6 8 }
* oddNumbers contains 5 elements: { 1 3 5 7 9 }
* numbers UnionWith oddNumbers...
* numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
*/

```

Remarks

This method is an O(n) operation, where n is the number of elements in the `other` parameter.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.ICollection<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to an [ICollection<T>](#) object.

C#

```
void ICollection<T>.Add(T item);
```

Parameters

item T

The object to add to the [ICollection<T>](#) object.

Implements

[Add\(T\)](#)

Exceptions

[NotSupportedException](#)

The [ICollection<T>](#) is read-only.

Remarks

If [Count](#) is less than [Capacity](#), this method is an O(1) operation. If the capacity must be increased to accommodate the new element, this method becomes an O([n](#)) operation, where [n](#) is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.ICollection<T>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether a collection is read-only.

C#

```
bool System.Collections.Generic.ICollection<T>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the collection is read-only; otherwise, `false`.

Implements

[IsReadOnly](#)

Remarks

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

HashSet<T>.IEnumerable<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerator<T> IEnumerable<T>.GetEnumerator();
```

Returns

[IEnumerator<T>](#)

An [IEnumerator<T>](#) object that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator object instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and its behavior is undefined.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

HashSet<T>.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) object that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and its behavior is undefined.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator](#)

IAsyncEnumerable<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Exposes an enumerator that provides asynchronous iteration over values of a specified type.

C#

```
public interface IAsyncEnumerable<out T>
```

Type Parameters

T

The type of values to enumerate.

This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Derived [System.Linq.IOrderedEnumerable<TElement>](#)

Methods

[+] [Expand table](#)

GetAsyncEnumerator(Cancellation Token)	Returns an enumerator that iterates asynchronously through the collection.
--	--

Extension Methods

[+] [Expand table](#)

ConfigureAwait<T>(IAsyncEnumerable<T>, Boolean)	Configures how awaits on the tasks returned from an async iteration will be performed.
WithCancellation<T>(IAsyncEnumerable<T>, CancellationToken)	Sets the CancellationToken to be passed to GetAsyncEnumerator(CancellationToken) when iterating.

Applies to

Product	Versions
.NET	Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.0 (package-provided), 2.1

IAsyncEnumerable<T>.GetAsyncEnumerator(CancellationToken) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Returns an enumerator that iterates asynchronously through the collection.

C#

```
public System.Collections.Generic.IAsyncEnumerator<out T>
GetAsyncEnumerator(System.Threading.CancellationToken cancellationToken =
default);
```

Parameters

cancellationToken [CancellationToken](#)

A [CancellationToken](#) that may be used to cancel the asynchronous iteration.

Returns

[IAsyncEnumerator<T>](#)

An enumerator that can be used to iterate asynchronously through the collection.

Applies to

Product	Versions
.NET	Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.6.2 (package-provided), 4.7 (package-provided)
.NET Standard	2.0 (package-provided), 2.1

IAsyncEnumerator<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Supports a simple asynchronous iteration over a generic collection.

C#

```
public interface IAsyncEnumerator<out T> : IAsyncDisposable
```

Type Parameters

T

The type of objects to enumerate.

This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Implements [IAsyncDisposable](#)

Properties

 [Expand table](#)

Current	Gets the element in the collection at the current position of the enumerator.
-------------------------	---

Methods

 [Expand table](#)

Dispose Async()	Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources asynchronously. (Inherited from IAsyncDisposable)
MoveNext Async()	Advances the enumerator asynchronously to the next element of the collection.

Extension Methods

 [Expand table](#)

<code>ConfigureAwait(IAsyncDisposable, Boolean)</code>	Configures how awaits on the tasks returned from an async disposable will be performed.
--	---

Applies to

Product	Versions
.NET	Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.6.2 (package-provided), 4.7 (package-provided)
.NET Standard	2.0 (package-provided), 2.1

IAsyncEnumerator<T>.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the element in the collection at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the collection at the current position of the enumerator.

Applies to

Product	Versions
.NET	Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.6.2 (package-provided), 4.7 (package-provided)
.NET Standard	2.0 (package-provided), 2.1

IAsyncEnumerator<T>.MoveNextAsync Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Advances the enumerator asynchronously to the next element of the collection.

C#

```
public System.Threading.Tasks.ValueTask<bool> MoveNextAsync();
```

Returns

[ValueTask<Boolean>](#)

A [ValueTask<TResult>](#) that will complete with a result of `true` if the enumerator was successfully advanced to the next element, or `false` if the enumerator has passed the end of the collection.

Applies to

Product	Versions
.NET	Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.0 (package-provided), 2.1

`I`Collection`<T>` Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Defines methods to manipulate generic collections.

C#

```
public interface ICollection<T> : System.Collections.Generic.IEnumerable<T>
```

Type Parameters

T

The type of the elements in the collection.

Derived [Microsoft.Extensions.AI.AdditionalPropertiesDictionary< TValue >](#)

[Microsoft.Extensions.AI.FunctionArguments](#)

[Microsoft.Extensions.AI.GeneratedEmbeddings< TEmbedding >](#)

[Microsoft.Extensions.DependencyInjection.IServiceCollection](#)

[More...](#)

Implements [IEnumerable< T >](#), [IEnumerable](#)

Examples

The following example implements the `I`Collection`<T>` interface to create a collection of custom `Box` objects named `BoxCollection`. Each `Box` has height, length, and width properties, which are used to define equality. Equality can be defined as all dimensions being the same or the volume being the same. The `Box` class implements the `IEquatable< T >` interface to define the default equality as the dimensions being the same.

The `BoxCollection` class implements the `Contains` method to use the default equality to determine whether a `Box` is in the collection. This method is used by the `Add` method so that each `Box` added to the collection has a unique set of dimensions. The `BoxCollection` class also provides an overload of the `Contains` method that takes a specified `EqualityComparer< T >` object, such as `BoxSameDimensions` and `BoxSameVol` classes in the example.

This example also implements an `IEnumerator<T>` interface for the `BoxCollection` class so that the collection can be enumerated.

C#

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

class Program
{
    static void Main(string[] args)
    {
        BoxCollection bxList = new BoxCollection();

        bxList.Add(new Box(10, 4, 6));
        bxList.Add(new Box(4, 6, 10));
        bxList.Add(new Box(6, 10, 4));
        bxList.Add(new Box(12, 8, 10));

        // Same dimensions. Cannot be added:
        bxList.Add(new Box(10, 4, 6));

        // Test the Remove method.
        Display(bxList);
        Console.WriteLine("Removing 6x10x4");
        bxList.Remove(new Box(6, 10, 4));
        Display(bxList);

        // Test the Contains method.
        Box BoxCheck = new Box(8, 12, 10);
        Console.WriteLine("Contains {0}x{1}x{2} by dimensions: {3}",
            BoxCheck.Height.ToString(), BoxCheck.Length.ToString(),
            BoxCheck.Width.ToString(), bxList.Contains(BoxCheck).ToString());

        // Test the Contains method overload with a specified equality comparer.
        Console.WriteLine("Contains {0}x{1}x{2} by volume: {3}",
            BoxCheck.Height.ToString(), BoxCheck.Length.ToString(),
            BoxCheck.Width.ToString(), bxList.Contains(BoxCheck,
            new BoxSameVol()).ToString());
    }

    public static void Display(BoxCollection bxList)
    {
        Console.WriteLine("\nHeight\tLength\tWidth\tHash Code");
        foreach (Box bx in bxList)
        {
            Console.WriteLine("{0}\t{1}\t{2}\t{3}",
                bx.Height.ToString(), bx.Length.ToString(),
                bx.Width.ToString(), bx.GetHashCode().ToString());
        }
    }

    // Results by manipulating the enumerator directly:
```

```
//IEnumerator enumerator = bxList.GetEnumerator();
//Console.WriteLine("\nHeight\tLength\tWidth\tHash Code");
//while (enumerator.MoveNext())
//{
//    Box b = (Box)enumerator.Current;
//    Console.WriteLine("{0}\t{1}\t{2}\t{3}",
//        b.Height.ToString(), b.Length.ToString(),
//        b.Width.ToString(), b.GetHashCode().ToString());
//}

Console.WriteLine();
}

}

public class Box : IEquatable<Box>
{
    public Box(int h, int l, int w)
    {
        this.Height = h;
        this.Length = l;
        this.Width = w;
    }
    public int Height { get; set; }
    public int Length { get; set; }
    public int Width { get; set; }

    // Defines equality using the
    // BoxSameDimensions equality comparer.
    public bool Equals(Box other)
    {
        if (new BoxSameDimensions().Equals(this, other))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public override bool Equals(object obj)
    {
        return base.Equals(obj);
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}

public class BoxCollection : ICollection<Box>
```

```
{  
    // The generic enumerator obtained from I Enumerator<Box>  
    // by Get Enumerator can also be used with the non-generic I Enumerator.  
    // To avoid a naming conflict, the non-generic I Enumerable method  
    // is explicitly implemented.  
  
    public I Enumerator<Box> Get Enumerator()  
    {  
        return new Box Enumerator(this);  
    }  
    I Enumerator I Enumerable.Get Enumerator()  
    {  
        return new Box Enumerator(this);  
    }  
  
    // The inner collection to store objects.  
    private List<Box> inner Col;  
  
    public Box Collection()  
    {  
        inner Col = new List<Box>();  
    }  
  
    // Adds an index to the collection.  
    public Box this[int index]  
    {  
        get { return (Box)inner Col[index]; }  
        set { inner Col[index] = value; }  
    }  
  
    // Determines if an item is in the collection  
    // by using the Box Same Dimensions equality comparer.  
    public bool Contains(Box item)  
    {  
        bool found = false;  
  
        foreach (Box bx in inner Col)  
        {  
            // Equality defined by the Box  
            // class's implementation of IEquatable<T>.  
            if (bx.Equals(item))  
            {  
                found = true;  
            }  
        }  
  
        return found;  
    }  
  
    // Determines if an item is in the  
    // collection by using a specified equality comparer.  
    public bool Contains(Box item, EqualityComparer<Box> comp)  
    {  
        bool found = false;
```

```
        foreach (Box bx in innerCol)
    {
        if (comp.Equals(bx, item))
        {
            found = true;
        }
    }

    return found;
}

// Adds an item if it is not already in the collection
// as determined by calling the Contains method.
public void Add(Box item)
{
    if (!Contains(item))
    {
        innerCol.Add(item);
    }
    else
    {
        Console.WriteLine("A box with {0}x{1}x{2} dimensions was already added
to the collection.",
                           item.Height.ToString(), item.Length.ToString(),
item.Width.ToString());
    }
}

public void Clear()
{
    innerCol.Clear();
}

public void CopyTo(Box[] array, int arrayIndex)
{
    if (array == null)
        throw new ArgumentNullException("The array cannot be null.");
    if (arrayIndex < 0)
        throw new ArgumentOutOfRangeException("The starting array index cannot
be negative.");
    if (Count > array.Length - arrayIndex)
        throw new ArgumentException("The destination array has fewer elements
than the collection.");

    for (int i = 0; i < innerCol.Count; i++)
        array[i + arrayIndex] = innerCol[i];
}

public int Count
{
    get
    {
        return innerCol.Count;
    }
}
```

```
        }

    }

    public bool IsReadOnly
    {
        get { return false; }
    }

    public bool Remove(Box item)
    {
        bool result = false;

        // Iterate the inner collection to
        // find the box to be removed.
        for (int i = 0; i < innerCol.Count; i++)
        {

            Box curBox = (Box)innerCol[i];

            if (new BoxSameDimensions().Equals(curBox, item))
            {
                innerCol.RemoveAt(i);
                result = true;
                break;
            }
        }
        return result;
    }
}

// Defines the enumerator for the Boxes collection.
// (Some prefer this class nested in the collection class.)
public class BoxEnumerator : IEnumerator<Box>
{
    private BoxCollection _collection;
    private int curIndex;
    private Box curBox;

    public BoxEnumerator(BoxCollection collection)
    {
        _collection = collection;
        curIndex = -1;
        curBox = default(Box);
    }

    public bool MoveNext()
    {
        //Avoids going beyond the end of the collection.
        if (++curIndex >= _collection.Count)
        {
            return false;
        }
        else
        {
```

```

        // Set current box to next item in collection.
        curBox = _collection[curIndex];
    }
    return true;
}

public void Reset() { curIndex = -1; }

void IDisposable.Dispose() { }

public Box Current
{
    get { return curBox; }
}

object IEnumerator.Current
{
    get { return Current; }
}
}

// Defines two boxes as equal if they have the same dimensions.
public class BoxSameDimensions : EqualityComparer<Box>
{
    public override bool Equals(Box b1, Box b2)
    {
        if (b1.Height == b2.Height && b1.Length == b2.Length
            && b1.Width == b2.Width)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode(Box bx)
    {
        int hCode = bx.Height ^ bx.Length ^ bx.Width;
        return hCode.GetHashCode();
    }
}

// Defines two boxes as equal if they have the same volume.
public class BoxSameVol : EqualityComparer<Box>
{
    public override bool Equals(Box b1, Box b2)
    {
        if ((b1.Height * b1.Length * b1.Width) ==
            (b2.Height * b2.Length * b2.Width))
        {
            return true;
        }
    }
}
```

```

        }
    else
    {
        return false;
    }
}

public override int GetHashCode(Box bx)
{
    int hCode = bx.Height ^ bx.Length ^ bx.Width;
    Console.WriteLine("HC: {0}", hCode.GetHashCode());
    return hCode.GetHashCode();
}
}

```

/*
This code example displays the following output:
=====

A box with 10x4x6 dimensions was already added to the collection.

Height	Length	Width	Hash Code
10	4	6	46104728
4	6	10	12289376
6	10	4	43495525
12	8	10	55915408

Removing 6x10x4

Height	Length	Width	Hash Code
10	4	6	46104728
4	6	10	12289376
12	8	10	55915408

Contains 8x12x10 by dimensions: False
Contains 8x12x10 by volume: True
*/

Remarks

The [ICollection<T>](#) interface is the base interface for classes in the [System.Collections.Generic](#) namespace.

The [ICollection<T>](#) interface extends [IEnumerable<T>](#); [IDictionary<TKey,TValue>](#) and [IList<T>](#) are more specialized interfaces that extend [ICollection<T>](#). A [IDictionary<TKey,TValue>](#) implementation is a collection of key/value pairs, like the [Dictionary<TKey,TValue>](#) class. A [IList<T>](#) implementation is a collection of values, and its members can be accessed by index, like the [List<T>](#) class.

If neither the [IDictionary<TKey,TValue>](#) interface nor the [IList<T>](#) interface meet the requirements of the required collection, derive the new collection class from the [ICollection<T>](#) interface instead for more flexibility.

Properties

[+] [Expand table](#)

Count	Gets the number of elements contained in the ICollection<T> .
IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.

Methods

[+] [Expand table](#)

Add(T)	Adds an item to the ICollection<T> .
Clear()	Removes all items from the ICollection<T> .
Contains(T)	Determines whether the ICollection<T> contains a specific value.
CopyTo(T[], Int32)	Copies the elements of the ICollection<T> to an Array , starting at a particular Array index.
GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)
Remove(T)	Removes the first occurrence of a specific object from the ICollection<T> .

Extension Methods

[+] [Expand table](#)

ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a

	transformation function to the source keys.
TolImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer< TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
TolImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer< TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
TolImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
TolImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
TolImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.
TolImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
TolImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer< TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
TolImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer< TKey>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
TolImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.

<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.

<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the average of a sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.

	transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most <code>size</code> .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.

<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.

<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.

<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the

	default equality comparer to compare values.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on matching keys. A specified IEqualityComparer<T> is used to compare keys.
Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>)	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the last element of a sequence that satisfies a specified condition.
Last<TSource>(IEnumerable<TSource>)	Returns the last element of a sequence.
LastOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
LastOrDefault<TSource>(IEnumerable<TSource>)	Returns the last element of a sequence, or a default value if the sequence contains no elements.
LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns an Int64 that represents how many elements in a sequence satisfy a condition.

<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.
<code>OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.

<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEqualityComparer<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing

	their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then

	returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is

	true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.

ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.

InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionary<TKey, TValue>](#)
- [IList<T>](#)
- [System.Collections](#)

ICollection<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the number of elements contained in the [ICollection<T>](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [ICollection<T>](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ICollection<T>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
public bool IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`.

Remarks

A collection that is read-only does not allow the addition or removal of elements after the collection is created. Note that read-only in this context does not indicate whether individual elements of the collection can be modified, since the [ICollection<T>](#) interface only supports addition and removal operations. For example, the [IsReadOnly](#) property of an array that is cast or converted to an [ICollection<T>](#) object returns `true`, even though individual array elements can be modified.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ICollection<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Adds an item to the [ICollection<T>](#).

C#

```
public void Add(T item);
```

Parameters

item [T](#)

The object to add to the [ICollection<T>](#).

Exceptions

[NotSupportedException](#)

The [ICollection<T>](#) is read-only.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

ICollection<T>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Removes all items from the [ICollection<T>](#).

C#

```
public void Clear();
```

Exceptions

[NotSupportedException](#)

The [ICollection<T>](#) is read-only.

Remarks

[Count](#) must be set to 0, and references to other objects from elements of the collection must be released.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

ICollection<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the [ICollection<T>](#) contains a specific value.

C#

```
public bool Contains(T item);
```

Parameters

item T

The object to locate in the [ICollection<T>](#).

Returns

Boolean

`true` if `item` is found in the [ICollection<T>](#); otherwise, `false`.

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Comparer<T>.Default](#), whereas [Dictionary<TKey,TValue>](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ICollection<T>.CopyTo(T[], Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Copies the elements of the [ICollection<T>](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
public void CopyTo(T[] array, int arrayIndex);
```

Parameters

array `T[]`

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection<T>](#).
The [Array](#) must have zero-based indexing.

arrayIndex `Int32`

The zero-based index in [array](#) at which copying begins.

Exceptions

[ArgumentNullException](#)

[array](#) is `null`.

[ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than 0.

[ArgumentException](#)

The number of elements in the source [ICollection<T>](#) is greater than the available space from [arrayIndex](#) to the end of the destination [array](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ICollection<T>.Remove(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Removes the first occurrence of a specific object from the [ICollection<T>](#).

C#

```
public bool Remove(T item);
```

Parameters

item T

The object to remove from the [ICollection<T>](#).

Returns

Boolean

`true` if `item` was successfully removed from the [ICollection<T>](#); otherwise, `false`. This method also returns `false` if `item` is not found in the original [ICollection<T>](#).

Exceptions

[NotSupportedException](#)

The [ICollection<T>](#) is read-only.

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Comparer<T>.Default](#), whereas, [Dictionary<TKey,TValue>](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

In collections of contiguous elements, such as lists, the elements that follow the removed element move up to occupy the vacated spot. If the collection is indexed, the indexes of the elements that are moved are also updated. This behavior does not apply to collections where elements are conceptually grouped into buckets, such as a hash table.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

IComparer<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Defines a method that a type implements to compare two objects.

C#

```
public interface IComparer<in T>
```

Type Parameters

T

The type of objects to compare.

This type parameter is contravariant. That is, you can use either the type you specified or any type that is less derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Derived [Microsoft.Extensions.Configuration.ConfigurationKeyComparer](#)

[Microsoft.Extensions.Primitives.StringSegmentComparer](#)

[System.Collections.Generic.Comparer<T>](#)

[System.Numerics.TotalOrderleee754Comparer<T>](#)

[System.Reflection.Metadata.HandleComparer](#)

[More...](#)

Examples

The following example implements the `IComparer<T>` interface to compare objects of type `Box` according to their dimensions. This example is part of a larger example provided for the `Comparer<T>` class.

C#

```
// This class is not demonstrated in the Main method
// and is provided only to show how to implement
// the interface. It is recommended to derive
// from Comparer<T> instead of implementing IComparer<T>.
public class BoxComp : IComparer<Box>
{
```

```
// Compares by Height, Length, and Width.
public int Compare(Box x, Box y)
{
    if (x.Height.CompareTo(y.Height) != 0)
    {
        return x.Height.CompareTo(y.Height);
    }
    else if (x.Length.CompareTo(y.Length) != 0)
    {
        return x.Length.CompareTo(y.Length);
    }
    else if (x.Width.CompareTo(y.Width) != 0)
    {
        return x.Width.CompareTo(y.Width);
    }
    else
    {
        return 0;
    }
}
```

Remarks

This interface is used with the [List<T>.Sort](#) and [List<T>.BinarySearch](#) methods. It provides a way to customize the sort order of a collection. Classes that implement this interface include the [SortedDictionary< TKey, TValue >](#) and [SortedList< TKey, TValue >](#) generic classes.

The default implementation of this interface is the [Comparer<T>](#) class. The [StringComparer](#) class implements this interface for type [String](#).

This interface supports ordering comparisons. That is, when the [Compare](#) method returns 0, it means that two objects sort the same. Implementation of exact equality comparisons is provided by the [IEqualityComparer<T>](#) generic interface.

We recommend that you derive from the [Comparer<T>](#) class instead of implementing the [IComparer<T>](#) interface, because the [Comparer<T>](#) class provides an explicit interface implementation of the [IComparer.Compare](#) method and the [Default](#) property that gets the default comparer for the object.

Methods

 [Expand table](#)

Compare(T, T)	Compares two objects and returns a value indicating whether one is less than, equal to, or greater than the other.
-------------------------------	--

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Comparer<T>](#)
- [System.Collections](#)
- [CurrentCulture](#)
- [CompareInfo](#)
- [CultureInfo](#)
- [IEqualityComparer<T>](#)

IComparer<T>.Compare(T, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Compares two objects and returns a value indicating whether one is less than, equal to, or greater than the other.

C#

```
public int Compare(T? x, T? y);
```

Parameters

x **T**

The first object to compare.

y **T**

The second object to compare.

Returns

Int32

A signed integer that indicates the relative values of **x** and **y**, as shown in the following table.

[] [Expand table](#)

Value	Meaning
Less than zero	x is less than y .
Zero	x equals y .
Greater than zero	x is greater than y .

Examples

The following example implements the [IComparer<T>](#) interface to compare objects of type **Box** according to their dimensions. This example is part of a larger example provided for the

Comparer<T> class.

C#

```
// This class is not demonstrated in the Main method
// and is provided only to show how to implement
// the interface. It is recommended to derive
// from Comparer<T> instead of implementing IComparer<T>.
public class BoxComp : IComparer<Box>
{
    // Compares by Height, Length, and Width.
    public int Compare(Box x, Box y)
    {
        if (x.Height.CompareTo(y.Height) != 0)
        {
            return x.Height.CompareTo(y.Height);
        }
        else if (x.Length.CompareTo(y.Length) != 0)
        {
            return x.Length.CompareTo(y.Length);
        }
        else if (x.Width.CompareTo(y.Width) != 0)
        {
            return x.Width.CompareTo(y.Width);
        }
        else
        {
            return 0;
        }
    }
}
```

Remarks

Implement this method to provide a customized sort order comparison for type `T`.

Comparing `null` with any reference type is allowed and does not generate an exception. A null reference is considered to be less than any reference that is not null.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [IComparable<T>](#)
- [CurrentCulture](#)
- [CompareInfo](#)
- [CultureInfo](#)

IDictionary<TKey,TValue> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Represents a generic collection of key/value pairs.

C#

```
public interface IDictionary<TKey, TValue> :  
    System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKe  
y, TValue>>,  
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKe  
y, TValue>>
```

Type Parameters

TKey

The type of keys in the dictionary.

TValue

The type of values in the dictionary.

Derived [Microsoft.Extensions.AI.AdditionalPropertiesDictionary<TValue>](#)

[Microsoft.Extensions.AI.AIFunctionArguments](#)

[System.Activities.Hosting.SymbolResolver](#)

[System.Activities.Presentation.Model.ModellItemDictionary](#)

[System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>](#)

[More...](#)

Implements [ICollection<KeyValuePair<TKey,TValue>>](#),

[IEnumerable<KeyValuePair<TKey,TValue>>](#), [IEnumerable<T>](#), [IEnumerable](#)

Examples

The following code example creates an empty [Dictionary<TKey,TValue>](#) of strings, with string keys, and accesses it through the [IDictionary<TKey,TValue>](#) interface.

The code example uses the [Add](#) method to add some elements. The example demonstrates that the [Add](#) method throws [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 how to use the [ContainsKey](#) method to test whether a key exists prior to calling the [Add](#) method.

Finally, the example shows how to enumerate the keys and values in the dictionary, and how to enumerate the values alone using the [Values](#) property.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys,
        // and access it through the IDictionary generic interface.
        IDictionary<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.
ICollection<string> icoll = openWith.Values;

```

```

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

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

// The elements of theValueCollection are strongly typed
// with the type that was specified for dictionary values.
Console.WriteLine();
foreach( string s in icoll )
{
    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.
*/
```

Remarks

The [IDictionary<TKey,TValue>](#) interface is the base interface for generic collections of key/value pairs.

Each element is a key/value pair stored in a [KeyValuePair<TKey,TValue>](#) object.

Each pair must have a unique key. Implementations can vary in whether they allow `key` to be `null`. The value can be `null` and does not have to be unique. The [IDictionary<TKey,TValue>](#) interface allows the contained keys and values to be enumerated, but it does not imply any particular sort order.

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 [IDictionary<TKey,TValue>](#) 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 [KeyValuePair<TKey,TValue>](#). For example:

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

The `foreach` statement is a wrapper around the enumerator, which only allows reading from, not writing to, the collection.

! Note

Because keys can be inherited and their behavior changed, their absolute uniqueness cannot be guaranteed by comparisons using the [Equals](#) method.

Notes to Implementers

The implementing class must have a means to compare keys.

Properties

 Expand table

Count	Gets the number of elements contained in the ICollection<T> . (Inherited from ICollection<T>)
IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only. (Inherited from ICollection<T>)
Item[TKey]	Gets or sets the element with the specified key.
Keys	Gets an ICollection<T> containing the keys of the IDictionary<TKey,TValue> .
Values	Gets an ICollection<T> containing the values in the IDictionary<TKey,TValue> .

Methods

 Expand table

Add(T)	Adds an item to the ICollection<T> . (Inherited from ICollection<T>)
Add(TKey, TValue)	Adds an element with the provided key and value to the IDictionary<TKey,TValue> .
Clear()	Removes all items from the ICollection<T> . (Inherited from ICollection<T>)
Contains(T)	Determines whether the ICollection<T> contains a specific value. (Inherited from ICollection<T>)
ContainsKey(TKey)	Determines whether the IDictionary<TKey,TValue> contains an element with the specified key.
CopyTo(T[], Int32)	Copies the elements of the ICollection<T> to an Array , starting at a particular Array index. (Inherited from ICollection<T>)
GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)

Remove(TKey)	Removes the element with the specified key from the <code>IDictionary<TKey,TValue></code> .
TryGetValue(TKey, TValue)	Gets the value associated with the specified key.

Extension Methods

[\[+\] Expand table](#)

Remove<TKey,TValue>(IDictionary<TKey,TValue>, TKey, TValue)	Tries to remove the value with the specified <code>key</code> from the <code>dictionary</code> .
TryAdd<TKey,TValue>(IDictionary<TKey,TValue>, TKey, TValue)	Tries to add the specified <code>key</code> and <code>value</code> to the <code>dictionary</code> .
ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its

	contents, and uses the specified equality comparer for the set type.
ToImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.
ToImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T>

	object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .

<code>Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Concatenates two sequences.
<code>Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)</code>	Determines whether a sequence contains a specified element by using a specified <code>IEqualityComparer<T></code> .
<code>Contains<TSource>(IEnumerable<TSource>, TSource)</code>	Determines whether a sequence contains a specified element by using the default equality comparer.
<code>Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns a number that represents how many elements in the specified sequence satisfy a condition.
<code>Count<TSource>(IEnumerable<TSource>)</code>	Returns the number of elements in a sequence.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>)</code>	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.

<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition

	<p>or a default value if no such element is found.</p>
FirstOrDefault<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence, or a default value if the sequence contains no elements.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Groups the elements of a sequence according to a specified key selector function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are

	projected by using a specified function.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on key equality and groups the results. A specified IEqualityComparer<T> is used to compare keys.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set intersection of two sequences by using the specified IEqualityComparer<T> to compare values.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set intersection of two sequences by using the default equality comparer to compare values.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on matching keys. A specified IEqualityComparer<T> is used to compare keys.
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.

<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IQueryable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IQueryable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.

<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.

<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an <code>IEnumerable</code> based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.

<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified <code>IEqualityComparer<T></code> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
<code>Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
<code>Single<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.

SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's

	<p>index is used in the logic of the predicate function.</p>
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the sum of the sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.

	transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.
Take<TSource>(IEnumerable<TSource>, Range)	Returns a specified range of contiguous elements from a sequence.
TakeLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns elements from a sequence as long as a specified condition is true.
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource, TKey, TElement>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource, TElement>, IEqualityComparer<TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T>

<code>Comparer<TKey>()</code>	according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToList<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer and an element selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>TryGetNonEnumeratedCount<TSource>(IQueryable<TSource>, Int32)</code>	Attempts to determine the number of elements in a sequence without forcing an enumeration.
<code>Union<TSource>(IQueryable<TSource>, IQueryable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set union of two sequences by using a specified <code>IEqualityComparer<T></code> .
<code>Union<TSource>(IQueryable<TSource>, IQueryable<TSource>)</code>	Produces the set union of two sequences by using the default

	equality comparer.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>Where<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Filters a sequence of values based on a predicate.
<code>Where<TSource>(IQueryable<TSource>, Func<TSource,Int32,Boolean>)</code>	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
<code>Zip<TFirst,TSecond>(IQueryable<TFirst>, IQueryable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IQueryable<TFirst>, IQueryable<TSecond>, IQueryable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IQueryable<TFirst>, IQueryable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IQueryable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IQueryable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IQueryable)</code>	Converts an <code>IEnumerable</code> to an <code>IQueryable</code> .
<code>AsQueryable<TElement>(IQueryable<TElement>)</code>	Converts a generic <code>IEnumerable<T></code> to a generic <code>IQueryable<T></code> .
<code>Ancestors<T>(IQueryable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching <code>XName</code> are included in the collection.
<code>Ancestors<T>(IQueryable<T>)</code>	Returns a collection of elements that contains the ancestors of

	every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection<T>](#)
- [System.Collections](#)

IDictionary< TKey, TValue >.Item[TKey] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets or sets the element with the specified key.

C#

```
public TValue this[TKey key] { get; set; }
```

Parameters

key `TKey`

The key of the element to get or set.

Property Value

`TValue`

The element with the specified key.

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[KeyNotFoundException](#)

The property is retrieved and `key` is not found.

[NotSupportedException](#)

The property is set and the [IDictionary< TKey, TValue >](#) is read-only.

Examples

The following code 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 also 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.

This code is part of a larger example that can be compiled and executed. See [System.Collections.Generic.IDictionary< TKey, TValue >](#).

C#

```
// 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";
```

C#

```
// 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.");
}
```

C#

```
// 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.");  
}
```

Remarks

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[key]` (`myCollection(key)` in Visual Basic).

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue` in C# (`myCollection("myNonexistentKey") = myValue` in Visual Basic). However, if the specified key already exists in the dictionary, setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

Implementations can vary in how they determine equality of objects; for example, the [List<T>](#) class uses [Comparer<T>.Default](#), whereas the [Dictionary<TKey,TValue>](#) class allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

The C# language uses the `this` keyword to define the indexers instead of implementing the [Item\[\]](#) property. Visual Basic implements [Item\[\]](#) as a default property, which provides the same indexing functionality.

Implementations can vary in whether they allow `key` to be `null`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Add\(TKey, TValue\)](#)
- [ContainsKey\(TKey\)](#)
- [IsReadOnly](#)

IDictionary< TKey, TValue >.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets an [ICollection<T>](#) containing the keys of the [IDictionary< TKey, TValue >](#).

C#

```
public System.Collections.Generic.ICollection<TKey> Keys { get; }
```

Property Value

[ICollection< TKey >](#)

An [ICollection<T>](#) containing the keys of the object that implements [IDictionary< TKey, TValue >](#).

Examples

The following code example shows how to enumerate keys alone using the [Keys](#) property.

This code is part of a larger example that can be compiled and executed. See [System.Collections.Generic.IDictionary< TKey, TValue >](#).

C#

```
// To get the keys alone, use the Keys property.
icoll = openWith.Keys;

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

Remarks

The order of the keys in the returned [ICollection<T>](#) is unspecified, but it is guaranteed to be the same order as the corresponding values in the [ICollection<T>](#) returned by the [Values](#) property.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection<T>](#)

IDictionary< TKey, TValue >.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets an [ICollection< T >](#) containing the values in the [IDictionary< TKey, TValue >](#).

C#

```
public System.Collections.Generic.ICollection<TValue> Values { get; }
```

Property Value

[ICollection< TValue >](#)

An [ICollection< T >](#) containing the values in the object that implements [IDictionary< TKey, TValue >](#).

Examples

The following code example shows how to enumerate values alone using the [Values](#) property.

This code is part of a larger example that can be compiled and executed. See [System.Collections.Generic.IDictionary< TKey, TValue >](#).

C#

```
// To get the values alone, use the Values property.
ICollection<string> icoll = 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 icoll )
{
    Console.WriteLine("Value = {0}", s);
}
```

Remarks

The order of the values in the returned `ICollection<T>` is unspecified, but it is guaranteed to be the same order as the corresponding keys in the `ICollection<T>` returned by the `Keys` property.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection<T>](#)

IDictionary<TKey,TValue>.Add(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Adds an element with the provided key and value to the [IDictionary<TKey,TValue>](#).

C#

```
public void Add(TKey key, TValue value);
```

Parameters

key TKey

The object to use as the key of the element to add.

value TValue

The object to use as the value of the element to add.

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

An element with the same key already exists in the [IDictionary<TKey,TValue>](#).

[NotSupportedException](#)

The [IDictionary<TKey,TValue>](#) is read-only.

Examples

The following code example creates an empty [Dictionary<TKey,TValue>](#) of strings, with integer keys, and accesses it through the [IDictionary<TKey,TValue>](#) interface. The code example 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.

This code is part of a larger example that can be compiled and executed. See [System.Collections.Generic.IDictionary< TKey, TValue >](#).

C#

```
// Create a new dictionary of strings, with string keys,
// and access it through the IDictionary generic interface.
IDictionary<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.");
}
```

Remarks

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue` in C# (`myCollection("myNonexistentKey") = myValue` in Visual Basic). However, if the specified key already exists in the dictionary, setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

Implementations can vary in how they determine equality of objects; for example, the [List<T>](#) class uses [Comparer<T>.Default](#), whereas the [Dictionary< TKey, TValue >](#) class allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

Implementations can vary in whether they allow `key` to be `null`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Item\[TKey\]](#)
- [IsReadOnly](#)

IDictionary< TKey, TValue >.ContainsKey(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the [IDictionary< TKey, TValue >](#) contains an element with the specified key.

C#

```
public bool ContainsKey(TKey key);
```

Parameters

key [TKey](#)

The key to locate in the [IDictionary< TKey, TValue >](#).

Returns

[Boolean](#)

`true` if the [IDictionary< TKey, TValue >](#) contains an element with the key; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [ContainsKey](#) method to test whether a key exists prior to calling the [Add](#) method. It also shows how to use the [TryGetValue](#) method, which can be a more efficient way to retrieve values if a program frequently tries key values that are not in the dictionary. Finally, it shows how to insert items using [Item\[\]](#) property (the indexer in C#).

This code is part of a larger example that can be compiled and executed. See [System.Collections.Generic.IDictionary< TKey, TValue >](#).

C#

```
// 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"]);
}
```

C#

```
// 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.");
}
```

C#

```
// 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.");
}
```

Remarks

Implementations can vary in how they determine equality of objects; for example, the [List<T>](#) class uses [Comparer<T>.Default](#), whereas the [Dictionary<TKey,TValue>](#) class allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

Implementations can vary in whether they allow `key` to be `null`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IDictionary<TKey,TValue>.Remove(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Removes the element with the specified key from the [IDictionary<TKey,TValue>](#).

C#

```
public bool Remove(TKey key);
```

Parameters

key TKey

The key of the element to remove.

Returns

[Boolean](#)

`true` if the element is successfully removed; otherwise, `false`. This method also returns `false` if `key` was not found in the original [IDictionary<TKey,TValue>](#).

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[NotSupportedException](#)

The [IDictionary<TKey,TValue>](#) is read-only.

Examples

The following code example shows how to remove a key/value pair from a dictionary using the [Remove](#) method.

This code is part of a larger example that can be compiled and executed. See [System.Collections.Generic.IDictionary<TKey,TValue>](#).

C#

```
// 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.");  
}
```

Remarks

Implementations can vary in how they determine equality of objects; for example, the [List<T>](#) class uses [Comparer<T>.Default](#), whereas the [Dictionary<TKey,TValue>](#) class allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

IDictionary< TKey, TValue >.TryGetValue(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the value associated with the specified key.

C#

```
public bool TryGetValue(TKey key, out TValue value);
```

Parameters

key TKey

The key whose value to get.

value TValue

When this method returns, the value associated with the specified key, if the key is found; otherwise, the default value for the type of the **value** parameter. This parameter is passed uninitialized.

Returns

Boolean

true if the object that implements [IDictionary< TKey, TValue >](#) contains an element with the specified key; otherwise, **false**.

Exceptions

[ArgumentNullException](#)

key is **null**.

Examples

The example shows how to use the [TryGetValue](#) method to retrieve values. If a program frequently tries key values that are not in a dictionary, the [TryGetValue](#) method can be more

efficient than using the `Item[]` property (the indexer in C#), which throws exceptions when attempting to retrieve nonexistent keys.

This code is part of a larger example that can be compiled and executed. See [System.Collections.Generic.IDictionary<TKey,TValue>](#).

C#

```
// 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.");
}
```

C#

```
// 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.");
}
```

Remarks

This method combines the functionality of the [ContainsKey](#) method and the `Item[]` property.

If the key is not found, then the `value` parameter gets the appropriate default value for the type `TValue`; for example, zero (0) for integer types, `false` for Boolean types, and `null` for reference types.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [Item\[TKey\]](#)

IEnumerable<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Exposes the enumerator, which supports a simple iteration over a collection of a specified type.

C#

```
public interface IEnumerable<out T> : System.Collections.IEnumerable
```

Type Parameters

T

The type of objects to enumerate.

This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Derived [Microsoft.Extensions.AI.AdditionalPropertiesDictionary< TValue >](#)
[Microsoft.Extensions.AI.AIFunctionArguments](#)
[Microsoft.Extensions.AI.GeneratedEmbeddings< TEmbedding >](#)
[Microsoft.Extensions.Configuration.Memory.MemoryConfigurationProvider](#)
[Microsoft.Extensions.DependencyInjection.IServiceCollection](#)
[More...](#)

Implements [IEnumerable](#)

Examples

The following example demonstrates how to implement the [IEnumerable<T>](#) interface and how to use that implementation to create a LINQ query. When you implement [IEnumerable<T>](#), you must also implement [IEnumerator<T>](#) or, for C# only, you can use the [yield](#) keyword. Implementing [IEnumerator<T>](#) also requires [IDisposable](#) to be implemented, which you will see in this example.

C#

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

public class App
{
    // Exercise the Iterator and show that it's more
    // performant.
    public static void Main()
    {
        TestStreamReaderEnumerable();
        Console.WriteLine("---");
        TestReadingFile();
    }

    public static void TestStreamReaderEnumerable()
    {
        // Check the memory before the iterator is used.
        long memoryBefore = GC.GetTotalMemory(true);
        IEnumerable<String> stringsFound;
        // Open a file with the StreamReaderEnumerable and check for a string.
        try {
            stringsFound =
                from line in new StreamReaderEnumerable(@"c:\temp\tempFile.txt")
                where line.Contains("string to search for")
                select line;
            Console.WriteLine("Found: " + stringsFound.Count());
        }
        catch (FileNotFoundException) {
            Console.WriteLine(@"This example requires a file named
C:\temp\tempFile.txt.");
            return;
        }

        // Check the memory after the iterator and output it to the console.
        long memoryAfter = GC.GetTotalMemory(false);
        Console.WriteLine("Memory Used With Iterator = \t"
            + string.Format(((memoryAfter - memoryBefore) / 1000).ToString(), "\n")
+ "kb");
    }

    public static void TestReadingFile()
    {
        long memoryBefore = GC.GetTotalMemory(true);
        StreamReader sr;
        try {
            sr = File.OpenText("c:\\temp\\\\tempFile.txt");
        }
        catch (FileNotFoundException) {
            Console.WriteLine(@"This example requires a file named
C:\temp\tempFile.txt.");
            return;
        }
    }
}
```

```

}

// Add the file contents to a generic list of strings.
List<string> fileContents = new List<string>();
while (!sr.EndOfStream) {
    fileContents.Add(sr.ReadLine());
}

// Check for the string.
var stringsFound =
    from line in fileContents
    where line.Contains("string to search for")
    select line;

sr.Close();
Console.WriteLine("Found: " + stringsFound.Count());

// Check the memory after when the iterator is not used, and output it to
// the console.
long memoryAfter = GC.GetTotalMemory(false);
Console.WriteLine("Memory Used Without Iterator = \t" +
    string.Format(((memoryAfter - memoryBefore) / 1000).ToString(), "n") +
"kb");
}

// A custom class that implements IEnumerable(T). When you implement
// IEnumerable(T),
// you must also implement IEnumerable and IEnumerator(T).
public class StreamReaderEnumerable : IEnumerable<string>
{
    private string _filePath;
    public StreamReaderEnumerable(string filePath)
    {
        _filePath = filePath;
    }

    // Must implement GetEnumerator, which returns a new StreamReaderEnumerator.
    public IEnumerator<string> GetEnumerator()
    {
        return new StreamReaderEnumerator(_filePath);
    }

    // Must also implement IEnumerable.GetEnumerator, but implement as a private
    // method.
    private IEnumerator GetEnumerator1()
    {
        return this.GetEnumerator();
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator1();
    }
}

```

```
// When you implement IEnumerable(T), you must also implement IEnumerator(T),
// which will walk through the contents of the file one line at a time.
// Implementing IEnumerator(T) requires that you implement IEnumerator and
// IDisposable.
public class StreamReaderEnumerator : IEnumerator<string>
{
    private StreamReader _sr;
    public StreamReaderEnumerator(string filePath)
    {
        _sr = new StreamReader(filePath);
    }

    private string _current;
    // Implement the IEnumerator(T).Current publicly, but implement
    // IEnumerator.Current, which is also required, privately.
    public string Current
    {

        get
        {
            if (_sr == null || _current == null)
            {
                throw new InvalidOperationException();
            }

            return _current;
        }
    }

    private object Current1
    {

        get { return this.Current; }
    }

    object IEnumerator.Current
    {
        get { return Current1; }
    }

    // Implement MoveNext and Reset, which are required by IEnumerator.
    public bool MoveNext()
    {
        _current = _sr.ReadLine();
        if (_current == null)
            return false;
        return true;
    }

    public void Reset()
    {
        _sr.DiscardBufferData();
        _sr.BaseStream.Seek(0, SeekOrigin.Begin);
        _current = null;
    }
}
```

```

// Implement IDisposable, which is also implemented by IEnumerator(T).
private bool disposedValue = false;
public void Dispose()
{
    Dispose(disposing: true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    if (!this.disposedValue)
    {
        if (disposing)
        {
            // Dispose of managed resources.
        }
        _current = null;
        if (_sr != null) {
            _sr.Close();
            _sr.Dispose();
        }
    }
    this.disposedValue = true;
}

~StreamReaderEnumerator()
{
    Dispose(disposing: false);
}
}
// This example displays output similar to the following:
//      Found: 2
//      Memory Used With Iterator =      33kb
//      ---
//      Found: 2
//      Memory Used Without Iterator =  206kb

```

Remarks

`IEnumerable<T>` is the base interface for collections in the `System.Collections.Generic` namespace such as `List<T>`, `Dictionary<TKey,TValue>`, and `Stack<T>` and other generic collections such as `ObservableCollection<T>` and `ConcurrentStack<T>`. Collections that implement `IEnumerable<T>` can be enumerated by using the `foreach` statement.

For the non-generic version of this interface, see [System.Collections.IEnumerable](#).

`IEnumerable<T>` contains a single method that you must implement when implementing this interface; `GetEnumerator`, which returns an `IEnumerator<T>` object. The returned

`IEnumerator<T>` provides the ability to iterate through the collection by exposing a [Current](#) property.

Notes to Implementers

To remain compatible with methods that iterate non-generic collections, `IEnumerable<T>` implements [IEnumerable](#). This allows a generic collection to be passed to a method that expects an [IEnumerable](#) object.

Methods

[] [Expand table](#)

GetEnumerator()	Returns an enumerator that iterates through the collection.
---------------------------------	---

Extension Methods

[] [Expand table](#)

ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.

<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input

	<code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the average of a sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.

Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most <code>size</code> .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a

	specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set difference of two sequences according to a specified key selector function.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set difference of two sequences according to a specified key selector function.
First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element in a sequence that satisfies a specified condition.
First<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence.
FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.

<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.

<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.

<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic

	sequence and returns the maximum resulting value.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function.
Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and

	returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in descending order according to a key.
Prepend<TSource>(IEnumerable<TSource>, TSource)	Adds a value to the beginning of the sequence.

<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified <code>IEqualityComparer<T></code> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.

Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.

SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key

	selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to specified key selector and element selector functions.

TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .

Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.

[Remove<T>\(IEnumerable<T>\)](#)

Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)
- [System.Collections](#)
- [Walkthrough: Implementing IEnumerable\(Of T\) in Visual Basic](#)
- [Iterators \(C#\)](#)
- [Iterators \(Visual Basic\)](#)

IEnumerable<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Returns an enumerator that iterates through the collection.

C#

```
public System.Collections.Generic.IEnumerator<out T> GetEnumerator();
```

Returns

[IEnumerator<T>](#)

An enumerator that can be used to iterate through the collection.

Examples

The following example demonstrates how to implement the [IEnumerable<T>](#) interface and uses that implementation to create a LINQ query. When you implement [IEnumerable<T>](#), you must also implement [IEnumerator<T>](#) or, for C# only, you can use the [yield](#) keyword.

Implementing [IEnumerator<T>](#) also requires [IDisposable](#) to be implemented, which you will see in this example.

C#

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

public class App
{
    // Exercise the Iterator and show that it's more
    // performant.
    public static void Main()
    {
        TestStreamReaderEnumerable();
        Console.WriteLine(" --- ");
        TestReadingFile();
    }
}
```

```
public static void TestStreamReaderEnumerable()
{
    // Check the memory before the iterator is used.
    long memoryBefore = GC.GetTotalMemory(true);
    IEnumerable<String> stringsFound;
    // Open a file with the StreamReaderEnumerable and check for a string.
    try {
        stringsFound =
            from line in new StreamReaderEnumerable(@"c:\temp\tempFile.txt")
            where line.Contains("string to search for")
            select line;
        Console.WriteLine("Found: " + stringsFound.Count());
    }
    catch (FileNotFoundException) {
        Console.WriteLine(@"This example requires a file named
C:\temp\tempFile.txt.");
        return;
    }

    // Check the memory after the iterator and output it to the console.
    long memoryAfter = GC.GetTotalMemory(false);
    Console.WriteLine("Memory Used With Iterator = \t"
        + string.Format(((memoryAfter - memoryBefore) / 1000).ToString(), "\n")
+ "kb");
}

public static void TestReadingFile()
{
    long memoryBefore = GC.GetTotalMemory(true);
    StreamReader sr;
    try {
        sr = File.OpenText("c:\\temp\\tempFile.txt");
    }
    catch (FileNotFoundException) {
        Console.WriteLine(@"This example requires a file named
C:\temp\tempFile.txt.");
        return;
    }

    // Add the file contents to a generic list of strings.
    List<string> fileContents = new List<string>();
    while (!sr.EndOfStream) {
        fileContents.Add(sr.ReadLine());
    }

    // Check for the string.
    var stringsFound =
        from line in fileContents
        where line.Contains("string to search for")
        select line;

    sr.Close();
    Console.WriteLine("Found: " + stringsFound.Count());
```

```

        // Check the memory after when the iterator is not used, and output it to
        // the console.
        long memoryAfter = GC.GetTotalMemory(false);
        Console.WriteLine("Memory Used Without Iterator = \t" +
            string.Format(((memoryAfter - memoryBefore) / 1000).ToString(), "n") +
        "kb");
    }
}

// A custom class that implements IEnumerable(T). When you implement
// IEnumerable(T),
// you must also implement IEnumerable and IEnumerator(T).
public class StreamReaderEnumerable : IEnumerable<string>
{
    private string _filePath;
    public StreamReaderEnumerable(string filePath)
    {
        _filePath = filePath;
    }

    // Must implement GetEnumerator, which returns a new StreamReaderEnumerator.
    public IEnumerator<string> GetEnumerator()
    {
        return new StreamReaderEnumerator(_filePath);
    }

    // Must also implement IEnumerable.GetEnumerator, but implement as a private
    // method.
    private IEnumerator GetEnumerator1()
    {
        return this.GetEnumerator();
    }
    IEnumerable IEnumerable.GetEnumerator()
    {
        return GetEnumerator1();
    }
}

// When you implement IEnumerable(T), you must also implement IEnumerator(T),
// which will walk through the contents of the file one line at a time.
// Implementing IEnumerator(T) requires that you implement IEnumerator and
// IDisposable.
public class StreamReaderEnumerator : IEnumerator<string>
{
    private StreamReader _sr;
    public StreamReaderEnumerator(string filePath)
    {
        _sr = new StreamReader(filePath);
    }

    private string _current;
    // Implement the IEnumerator(T).Current publicly, but implement
    // IEnumerator.Current, which is also required, privately.
    public string Current
    {

```

```
get
{
    if (_sr == null || _current == null)
    {
        throw new InvalidOperationException();
    }

    return _current;
}
}

private object Current1
{
    get { return this.Current; }
}

object IEnumerator.Current
{
    get { return Current1; }
}

// Implement MoveNext and Reset, which are required by IEnumerator.
public bool MoveNext()
{
    _current = _sr.ReadLine();
    if (_current == null)
        return false;
    return true;
}

public void Reset()
{
    _sr.DiscardBufferData();
    _sr.BaseStream.Seek(0, SeekOrigin.Begin);
    _current = null;
}

// Implement IDisposable, which is also implemented by IEnumerator(T).
private bool disposedValue = false;
public void Dispose()
{
    Dispose(disposing: true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    if (!this.disposedValue)
    {
        if (disposing)
        {
            // Dispose of managed resources.
        }
    }
}
```

```

        _current = null;
        if (_sr != null) {
            _sr.Close();
            _sr.Dispose();
        }
    }

    this.disposedValue = true;
}

~StreamReaderEnumerator()
{
    Dispose(disposing: false);
}
}

// This example displays output similar to the following:
//      Found: 2
//      Memory Used With Iterator =      33kb
//      ---
//      Found: 2
//      Memory Used Without Iterator =  206kb

```

Remarks

The returned `IEnumerator<T>` provides the ability to iterate through the collection by exposing a `Current` property. You can use enumerators to read the data in a collection, but not to modify the collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, `Current` is undefined. Therefore, you must call the `MoveNext` method to advance the enumerator to the first element of the collection before reading the value of `Current`.

`Current` returns the same object until `MoveNext` is called again as `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this position, subsequent calls to `MoveNext` also return `false`. If the last call to `MoveNext` returned `false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

If changes are made to the collection, such as adding, modifying, or deleting elements, the behavior of the enumerator is undefined.

An enumerator does not have exclusive access to the collection so an enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is invalidated and you may get

unexpected results. Also, enumerating a collection is not a thread-safe procedure. To guarantee thread-safety, you should lock the collection during enumerator or implement synchronization on the collection.

Default implementations of collections in the [System.Collections.Generic](#) namespace aren't synchronized.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)
- [Walkthrough: Implementing IEnumerable\(Of T\) in Visual Basic](#)
- [Iterators \(C#\)](#)
- [Iterators \(Visual Basic\)](#)

IEnumerator<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Supports a simple iteration over a generic collection.

C#

```
public interface IEnumerator<out T> : IDisposable, System.Collections.IEnumerator
```

Type Parameters

T

The type of objects to enumerate.

This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Derived [Microsoft.Extensions.AI.AdditionalPropertiesDictionary< TValue >.Enumerator](#)

[Microsoft.Extensions.Primitives.StringTokenizer.Enumerator](#)

[Microsoft.Extensions.Primitives.StringValues.Enumerator](#)

[Microsoft.IO Enumeration.FileSystemEnumerator< TResult >](#)

[Microsoft.VisualBasic.StlClr.DequeEnumerator< TValue >](#)

[More...](#)

Implements [IEnumerator](#), [IDisposable](#)

Examples

The following example shows an implementation of the [IEnumerator< T >](#) interface for a collection class of custom objects. The custom object is an instance of the type [Box](#), and the collection class is [BoxCollection](#). This code example is part of a larger example provided for the [ICollection< T >](#) interface.

C#

```
// Defines the enumerator for the Boxes collection.
```

```

// (Some prefer this class nested in the collection class.)
public class BoxEnumerator : IEnumerator<Box>
{
    private BoxCollection _collection;
    private int curIndex;
    private Box curBox;

    public BoxEnumerator(BoxCollection collection)
    {
        _collection = collection;
        curIndex = -1;
        curBox = default(Box);
    }

    public bool MoveNext()
    {
        //Avois going beyond the end of the collection.
        if (++curIndex >= _collection.Count)
        {
            return false;
        }
        else
        {
            // Set current box to next item in collection.
            curBox = _collection[curIndex];
        }
        return true;
    }

    public void Reset() { curIndex = -1; }

    void IDisposable.Dispose() { }

    public Box Current
    {
        get { return curBox; }
    }

    object IEnumerator.Current
    {
        get { return Current; }
    }
}

```

Remarks

`IEnumerator<T>` is the base interface for all generic enumerators.

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, `Current` is undefined. Therefore, you must call `MoveNext` to advance the enumerator to the first element of the collection before reading the value of `Current`.

`Current` returns the same object until `MoveNext` is called. `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this position, subsequent calls to `MoveNext` also return `false`. If the last call to `MoveNext` returned `false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

The `Reset` method is provided for COM interoperability. It does not necessarily need to be implemented; instead, the implementer can simply throw a `NotSupportedException`. However, if you choose to do this, you should make sure no callers are relying on the `Reset` functionality.

If changes are made to the collection, such as adding, modifying, or deleting elements, the behavior of the enumerator is undefined.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the `System.Collections.Generic` namespace are not synchronized.

Notes to Implementers

Implementing this interface requires implementing the nongeneric `IEnumerator` interface. The `MoveNext()` and `Reset()` methods do not depend on `T`, and appear only on the nongeneric interface. The `Current` property appears on both interfaces, and has different return types. Implement the nongeneric `Current` property as an explicit interface implementation. This allows any consumer of the nongeneric interface to consume the generic interface.

In addition, `IEnumerator<T>` implements `IDisposable`, which requires you to implement the `Dispose()` method. This enables you to close database connections or release file handles or

similar operations when using other resources. If there are no additional resources to dispose of, provide an empty [Dispose\(\)](#) implementation.

Properties

 [Expand table](#)

Current	Gets the element in the collection at the current position of the enumerator.
-------------------------	---

Methods

 [Expand table](#)

Dispose()	Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources. (Inherited from IDisposable)
Move Next()	Advances the enumerator to the next element of the collection. (Inherited from IEnumerator)
Reset()	Sets the enumerator to its initial position, which is before the first element in the collection. (Inherited from IEnumerator)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerable<T>](#)
- [ICollection<T>](#)
- [System.Collections](#)

IEnumerator<T>.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the element in the collection at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the collection at the current position of the enumerator.

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element in the collection, immediately after the enumerator is created. [MoveNext](#) must be called to advance the enumerator to the first element of the collection before reading the value of [Current](#).
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

Notes to Implementers

Implementing this interface requires implementing the nongeneric [IEnumerator](#) interface. The [Current](#) property appears on both interfaces, and has different return types. Implement the nongeneric [Current](#) property as an explicit interface implementation. This allows any consumer of the nongeneric interface to consume the generic interface.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)

IEqualityComparer<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Defines methods to support the comparison of objects for equality.

C#

```
public interface IEqualityComparer<in T>
```

Type Parameters

T

The type of objects to compare.

This type parameter is contravariant. That is, you can use either the type you specified or any type that is less derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Derived [Microsoft.Extensions.Primitives.StringSegmentComparer](#)

[System.Collections.Generic.EqualityComparer<T>](#)

[System.Collections.Generic.ReferenceEqualityComparer](#)

[System.Data.DataRowComparer<TRow>](#)

[System.Numerics.TotalOrderleee754Comparer<T>](#)

[More...](#)

Examples

The following example adds custom `Box` objects to a dictionary collection. The `Box` objects are considered equal if their dimensions are the same.

C#

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

static class Example
{
    static void Main()
    {
```

```
BoxEqualityComparer comparer = new();

Dictionary<Box, string> boxes = new(comparer);

AddBox(new Box(4, 3, 4), "red");
AddBox(new Box(4, 3, 4), "blue");
AddBox(new Box(3, 4, 3), "green");

Console.WriteLine($"The dictionary contains {boxes.Count} Box objects.");

void AddBox(Box box, string name)
{
    try
    {
        boxes.Add(box, name);
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Unable to add {box}: {e.Message}");
    }
}
}

class Box
{
    public int Height { get; }
    public int Length { get; }
    public int Width { get; }

    public Box(int height, int length, int width)
    {
        Height = height;
        Length = length;
        Width = width;
    }

    public override string ToString() => $"({Height}, {Length}, {Width})";
}

class BoxEqualityComparer : IEqualityComparer<Box>
{
    public bool Equals(Box? b1, Box? b2)
    {
        if (ReferenceEquals(b1, b2))
            return true;

        if (b2 is null || b1 is null)
            return false;

        return b1.Height == b2.Height
            && b1.Length == b2.Length
            && b1.Width == b2.Width;
    }
}
```

```

    public int GetHashCode(Box box) => box.Height ^ box.Length ^ box.Width;
}

// The example displays the following output:
//      Unable to add (4, 3, 4): An item with the same key has already been added.
//      The dictionary contains 2 Box objects.

```

Remarks

This interface allows the implementation of customized equality comparison for collections. That is, you can create your own definition of equality for type `T`, and specify that this definition be used with a collection type that accepts the `IEqualityComparer<T>` generic interface. In the .NET Framework, constructors of the `Dictionary< TKey, TValue >` generic collection type accept this interface.

A default implementation of this interface is provided by the `Default` property of the `EqualityComparer<T>` generic class. The `StringComparer` class implements `IEqualityComparer<T>` of type `String`.

This interface supports only equality comparisons. Customization of comparisons for sorting and ordering is provided by the `IComparer<T>` generic interface.

We recommend that you derive from the `EqualityComparer<T>` class instead of implementing the `IEqualityComparer<T>` interface, because the `EqualityComparer<T>` class tests for equality using the `IEquatable<T>.Equals` method instead of the `Object.Equals` method. This is consistent with the `Contains`, `IndexOf`, `LastIndexOf`, and `Remove` methods of the `Dictionary< TKey, TValue >` class and other generic collections.

Methods

[+] Expand table

<code>Equals(T, T)</code>	Determines whether the specified objects are equal.
<code>GetHashCode(T)</code>	Returns a hash code for the specified object.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [EqualityComparer<T>](#)
- [Dictionary<TKey,TValue>](#)
- [Dictionary<TKey,TValue>](#)
- [IComparer<T>](#)

IEqualityComparer<T>.Equals(T, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the specified objects are equal.

C#

```
public bool Equals(T? x, T? y);
```

Parameters

x **T**

The first object of type **T** to compare.

y **T**

The second object of type **T** to compare.

Returns

[Boolean](#)

`true` if the specified objects are equal; otherwise, `false`.

Remarks

Implement this method to provide a customized equality comparison for type **T**.

Notes to Implementers

Implementations are required to ensure that if the `Equals(T, T)` method returns `true` for two objects **x** and **y**, then the value returned by the `GetHashCode(T)` method for **x** must equal the value returned for **y**.

The `Equals(T, T)` method is reflexive, symmetric, and transitive. That is, it returns `true` if used to compare an object with itself; `true` for two objects **x** and **y** if it is `true` for **y** and **x**; and `true` for two objects **x** and **z** if it is `true` for **x** and **y** and also `true` for **y** and **z**.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IEqualityComparer<T>.GetHashCode(T)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Returns a hash code for the specified object.

C#

```
public int GetHashCode(T obj);
```

Parameters

obj **T**

The [Object](#) for which a hash code is to be returned.

Returns

[Int32](#)

A hash code for the specified object.

Exceptions

[ArgumentNullException](#)

The type of **obj** is a reference type and **obj** is **null**.

Remarks

Implement this method to provide a customized hash code for type **T**, corresponding to the customized equality comparison provided by the [Equals](#) method.

Notes to Implementers

Implementations are required to ensure that if the [Equals\(T, T\)](#) method returns **true** for two objects **x** and **y**, then the value returned by the [GetHashCode\(T\)](#) method for **x** must equal the

value returned for `y`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetHashCode\(\)](#)
- [IHashCodeProvider](#)

IList<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Represents a collection of objects that can be individually accessed by index.

C#

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

Type Parameters

T

The type of elements in the list.

Derived [Microsoft.Extensions.AI.GeneratedEmbeddings<TEmbedding>](#)
[Microsoft.Extensions.DependencyInjection.IServiceCollection](#)
[Microsoft.Extensions.Primitives.StringValues](#)
[System.Activities.Presentation.Model.ModellItemCollection](#)
[More...](#)

Implements [ICollection<T>](#) , [IEnumerable<T>](#) , [IEnumerable](#)

Remarks

The [IList<T>](#) generic interface is a descendant of the [ICollection<T>](#) generic interface and is the base interface of all generic lists.

Properties

 [Expand table](#)

Count	Gets the number of elements contained in the ICollection<T> . (Inherited from ICollection<T>)
IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.

(Inherited from [ICollection<T>](#))

Item[Int32]	Gets or sets the element at the specified index.
-----------------------------	--

Methods

[\[+\] Expand table](#)

Add(T)	Adds an item to the ICollection<T> . (Inherited from ICollection<T>)
Clear()	Removes all items from the ICollection<T> . (Inherited from ICollection<T>)
Contains(T)	Determines whether the ICollection<T> contains a specific value. (Inherited from ICollection<T>)
CopyTo(T[], Int32)	Copies the elements of the ICollection<T> to an Array , starting at a particular Array index. (Inherited from ICollection<T>)
GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)
IndexOf(T)	Determines the index of a specific item in the IList<T> .
Insert(Int32, T)	Inserts an item to the IList<T> at the specified index.
Remove(T)	Removes the first occurrence of a specific object from the ICollection<T> . (Inherited from ICollection<T>)
RemoveAt(Int32)	Removes the IList<T> item at the specified index.

Extension Methods

[\[+\] Expand table](#)

ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a

	transformation function to the source keys.
TolImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer< TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
TolImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer< TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
TolImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
TolImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
TolImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.
TolImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
TolImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer< TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
TolImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer< TKey>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
TolImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.

<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.

<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the average of a sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.

	transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most <code>size</code> .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.

<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.

<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.

<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the

	default equality comparer to compare values.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on matching keys. A specified IEqualityComparer<T> is used to compare keys.
Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>)	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the last element of a sequence that satisfies a specified condition.
Last<TSource>(IEnumerable<TSource>)	Returns the last element of a sequence.
LastOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
LastOrDefault<TSource>(IEnumerable<TSource>)	Returns the last element of a sequence, or a default value if the sequence contains no elements.
LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns an Int64 that represents how many elements in a sequence satisfy a condition.

<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.
<code>OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.

<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEqualityComparer<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing

	their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then

	returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is

	true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.

ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.

InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection<T>](#)
- [System.Collections](#)

IList<T>.Item[Int32] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets or sets the element at the specified index.

C#

```
public T this[int index] { get; set; }
```

Parameters

index [Int32](#)

The zero-based index of the element to get or set.

Property Value

T

The element at the specified index.

Exceptions

[ArgumentOutOfRangeException](#)

index is not a valid index in the [IList<T>](#).

[NotSupportedException](#)

The property is set and the [IList<T>](#) is read-only.

Remarks

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[index]`.

The C# language uses the [this](#) keyword to define the indexers instead of implementing the [Item\[\]](#) property. Visual Basic implements [Item\[\]](#) as a [default property](#), which provides the same indexing functionality.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

IList<T>.IndexOf(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines the index of a specific item in the [IList<T>](#).

C#

```
public int IndexOf(T item);
```

Parameters

item [T](#)

The object to locate in the [IList<T>](#).

Returns

[Int32](#)

The index of **item** if found in the list; otherwise, -1.

Remarks

If an object occurs multiple times in the list, the [IndexOf](#) method always returns the first instance found.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IList<T>.Insert(Int32, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Inserts an item to the [IList<T>](#) at the specified index.

C#

```
public void Insert(int index, T item);
```

Parameters

index [Int32](#)

The zero-based index at which [item](#) should be inserted.

item [T](#)

The object to insert into the [IList<T>](#).

Exceptions

[ArgumentOutOfRangeException](#)

[index](#) is not a valid index in the [IList<T>](#).

[NotSupportedException](#)

The [IList<T>](#) is read-only.

Remarks

If [index](#) equals the number of items in the [IList<T>](#), then [item](#) is appended to the list.

In collections of contiguous elements, such as lists, the elements that follow the insertion point move down to accommodate the new element. If the collection is indexed, the indexes of the elements that are moved are also updated. This behavior does not apply to collections where elements are conceptually grouped into buckets, such as a hash table.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

IList<T>.RemoveAt(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Removes the [IList<T>](#) item at the specified index.

C#

```
public void RemoveAt(int index);
```

Parameters

index [Int32](#)

The zero-based index of the item to remove.

Exceptions

[ArgumentOutOfRangeException](#)

index is not a valid index in the [IList<T>](#).

[NotSupportedException](#)

The [IList<T>](#) is read-only.

Remarks

In collections of contiguous elements, such as lists, the elements that follow the removed element move up to occupy the vacated spot. If the collection is indexed, the indexes of the elements that are moved are also updated. This behavior does not apply to collections where elements are conceptually grouped into buckets, such as a hash table.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

IReadOnlyCollection<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Represents a strongly-typed, read-only collection of elements.

C#

```
public interface IReadOnlyCollection<out T> :  
    System.Collections.Generic.IEnumerable<out T>
```

Type Parameters

T

The type of the elements.

This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Derived [Microsoft.Extensions.AI.AdditionalPropertiesDictionary< TValue >](#)

[Microsoft.Extensions.AI.AIFunctionArguments](#)

[Microsoft.Extensions.AI.GeneratedEmbeddings< TEmbedding >](#)

[Microsoft.Extensions.Primitives.StringValues](#)

[System.ArraySegment< T >](#)

[More...](#)

Implements [IEnumerable< T >](#), [IEnumerable](#)

Properties

 [Expand table](#)

Count	Gets the number of elements in the collection.
-------	--

Methods

GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)
---------------------------------	--

Extension Methods

TolImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
TolImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
TolImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.

<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.

<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Cast<TResult>(IEnumerable)</code>	Casts the elements of an IEnumerable to the specified type.
<code>Chunk<TSource>(IEnumerable<TSource>, Int32)</code>	Splits the elements of a sequence into chunks of size at most <code>size</code> .
<code>Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Concatenates two sequences.
<code>Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)</code>	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .

Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IQueryable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IQueryable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IQueryable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IQueryable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IQueryable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IQueryable<TSource>, Int32)	Returns the element at a specified index in a sequence.

ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set difference of two sequences according to a specified key selector function.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set difference of two sequences according to a specified key selector function.
First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element in a sequence that satisfies a specified condition.
First<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence.
FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence, or a default value if the sequence contains no elements.

<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a

	<p>result value from each group and its key. The elements of each group are projected by using a specified function.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.

<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Decimal>)</code>	Invokes a transform function on each element of a sequence and

	returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Invokes a transform function on each element of a generic

	sequence and returns the minimum resulting value.
MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Sorts the elements of a sequence in descending order according to a key.
Prepend<TSource>(IEnumerable<TSource>, TSource)	Adds a value to the beginning of the sequence.
Reverse<TSource>(IEnumerable<TSource>)	Inverts the order of the elements in a sequence.
Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Projects each element of a sequence into a new form.
Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)	Projects each element of a sequence into a new form by incorporating the element's index.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> ,

	<p>and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.</p>
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.

<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>Skip<TSource>(IEnumerable<TSource>, Int32)</code>	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
<code>SkipLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Computes the sum of the sequence of <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.

ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToListAsync<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.

Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the

	descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyCollection<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the number of elements in the collection.

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyDictionary<TKey, TValue> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Represents a generic read-only collection of key/value pairs.

C#

```
public interface IReadOnlyDictionary<TKey, TValue> :  
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>>,  
    System.Collections.Generic.IReadOnlyCollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>
```

Type Parameters

TKey

The type of keys in the read-only dictionary.

TValue

The type of values in the read-only dictionary.

Derived [Microsoft.Extensions.AI.AdditionalPropertiesDictionary<TValue>](#)

[Microsoft.Extensions.AI.AIFunctionArguments](#)

[System.Collections.Concurrent.ConcurrentDictionary<TKey, TValue>](#)

[System.Collections.Frozen.FrozenDictionary<TKey, TValue>](#)

[System.Collections.Generic.Dictionary<TKey, TValue>](#)

[More...](#)

Implements [IEnumerable<KeyValuePair<TKey, TValue>>](#), [IEnumerable<T>](#),

[IReadOnlyCollection<KeyValuePair<TKey, TValue>>](#), [IEnumerable](#)

Remarks

Each element is a key/value pair that is stored in a [KeyValuePair<TKey, TValue>](#) object.

Each pair must have a unique key. Implementations can vary in whether they allow you to specify a key that is `null`. The value can be `null` and does not have to be unique. The [IReadOnlyDictionary<TKey,TValue>](#) interface allows the contained keys and values to be enumerated, but it does not imply any particular sort order.

The `foreach` statement of the C# language (`For Each` in Visual Basic) requires the type of each element in the collection. Because each element of the [IReadOnlyDictionary<TKey,TValue>](#) interface 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 [KeyValuePair<TKey,TValue>](#), as the following example illustrates.

C#

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

The `foreach` statement is a wrapper around the enumerator; it allows only reading from the collection, not writing to the collection.

Properties

[+] Expand table

Count	Gets the number of elements in the collection. (Inherited from IReadOnlyCollection<T>)
Item[TKey]	Gets the element that has the specified key in the read-only dictionary.
Keys	Gets an enumerable collection that contains the keys in the read-only dictionary.
Values	Gets an enumerable collection that contains the values in the read-only dictionary.

Methods

[+] Expand table

ContainsKey(TKey)	Determines whether the read-only dictionary contains an element that has the specified key.
GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)

<code>TryGetValue(TKey, TValue)</code>	Gets the value that is associated with the specified key.
--	---

[+] [Expand table](#)

<code>GetValueOrDefault<TKey,TValue>(IReadOnlyDictionary<TKey,TValue>, TKey, TValue)</code>	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
<code>GetValueOrDefault<TKey,TValue>(IReadOnlyDictionary<TKey,TValue>, TKey)</code>	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
<code>ToImmutableArray<TSource>(IEnumerable<TSource>)</code>	Creates an immutable array from the specified collection.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its

	contents, and uses the specified equality comparer for the set type.
ToImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.
ToImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T>

	object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .

<code>Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Concatenates two sequences.
<code>Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)</code>	Determines whether a sequence contains a specified element by using a specified <code>IEqualityComparer<T></code> .
<code>Contains<TSource>(IEnumerable<TSource>, TSource)</code>	Determines whether a sequence contains a specified element by using the default equality comparer.
<code>Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns a number that represents how many elements in the specified sequence satisfy a condition.
<code>Count<TSource>(IEnumerable<TSource>)</code>	Returns the number of elements in a sequence.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>)</code>	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.

<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition

	<p>or a default value if no such element is found.</p>
FirstOrDefault<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence, or a default value if the sequence contains no elements.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Groups the elements of a sequence according to a specified key selector function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are

	projected by using a specified function.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on key equality and groups the results. A specified IEqualityComparer<T> is used to compare keys.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set intersection of two sequences by using the specified IEqualityComparer<T> to compare values.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set intersection of two sequences by using the default equality comparer to compare values.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on matching keys. A specified IEqualityComparer<T> is used to compare keys.
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.

<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IQueryable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IQueryable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.

<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.

<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an <code>IEnumerable</code> based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.

<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
<code>Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
<code>Single<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.

SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's

	<p>index is used in the logic of the predicate function.</p>
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the sum of the sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.

	transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.
Take<TSource>(IEnumerable<TSource>, Range)	Returns a specified range of contiguous elements from a sequence.
TakeLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns elements from a sequence as long as a specified condition is true.
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource, TKey, TElement>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource, TElement>, IEqualityComparer<TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T>

<code>Comparer<TKey>()</code>	according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToList<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer and an element selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>TryGetNonEnumeratedCount<TSource>(IQueryable<TSource>, Int32)</code>	Attempts to determine the number of elements in a sequence without forcing an enumeration.
<code>Union<TSource>(IQueryable<TSource>, IQueryable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set union of two sequences by using a specified <code>IEqualityComparer<T></code> .
<code>Union<TSource>(IQueryable<TSource>, IQueryable<TSource>)</code>	Produces the set union of two sequences by using the default

	equality comparer.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Filters a sequence of values based on a predicate.
<code>Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an <code>IEnumerable</code> to an <code>IQueryable</code> .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic <code>IEnumerable<T></code> to a generic <code>IQueryable<T></code> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching <code>XName</code> are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of

	every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyDictionary<TKey, TValue>.Item[TKey] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the element that has the specified key in the read-only dictionary.

C#

```
public TValue this[TKey key] { get; }
```

Parameters

key TKey

The key to locate.

Property Value

TValue

The element that has the specified key in the read-only dictionary.

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[KeyNotFoundException](#)

The property is retrieved and `key` is not found.

Remarks

This property lets you access a specific element in the collection by using the following syntax:

`myCollection[key]` (`myCollection(key)` in Visual Basic).

Implementations can vary in how they determine the equality of objects: for example, the class that implements [IReadOnlyDictionary<TKey,TValue>](#) might use the [Comparer<T>.Default](#) property, or it might implement the [IComparer<T>](#) method.

Implementations can vary in whether they allow `key` to be `null`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyDictionary<TKey,TValue>.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets an enumerable collection that contains the keys in the read-only dictionary.

C#

```
public System.Collections.Generic.IEnumerable<TKey> Keys { get; }
```

Property Value

[IEnumerable<TKey>](#)

An enumerable collection that contains the keys in the read-only dictionary.

Remarks

The order of the keys in the enumerable collection is unspecified, but the implementation must guarantee that the keys are in the same order as the corresponding values in the enumerable collection that is returned by the [Values](#) property.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyDictionary<TKey,TValue>.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets an enumerable collection that contains the values in the read-only dictionary.

C#

```
public System.Collections.Generic.IEnumerable<TValue> Values { get; }
```

Property Value

[IEnumerable<TValue>](#)

An enumerable collection that contains the values in the read-only dictionary.

Remarks

The order of the values in the enumerable collection is unspecified, but the implementation must guarantee that the values are in the same order as the corresponding keys in the enumerable collection that is returned by the [Keys](#) property.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyDictionary<TKey, TValue>.ContainsKey(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the read-only dictionary contains an element that has the specified key.

C#

```
public bool ContainsKey(TKey key);
```

Parameters

key TKey

The key to locate.

Returns

Boolean

`true` if the read-only dictionary contains an element that has the specified key; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Remarks

Implementations can vary in how they determine the equality of objects; for example, the class that implements [IReadOnlyDictionary<TKey, TValue>](#) might use the [Comparer<T>.Default](#) property, or it might implement the [IComparer<T>](#) method.

Implementations can vary in whether they allow `key` to be `null`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyDictionary<TKey,TValue>.TryGetValue(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the value that is associated with the specified key.

C#

```
public bool TryGetValue(TKey key, out TValue value);
```

Parameters

key TKey

The key to locate.

value TValue

When this method returns, the value associated with the specified key, if the key is found; otherwise, the default value for the type of the **value** parameter. This parameter is passed uninitialized.

Returns

Boolean

true if the object that implements the [IReadOnlyDictionary<TKey,TValue>](#) interface contains an element that has the specified key; otherwise, **false**.

Exceptions

[ArgumentNullException](#)

key is **null**.

Remarks

This method combines the functionality of the [ContainsKey](#) method and the [Item\[\]](#) property.

If the key is not found, the `value` parameter gets the appropriate default value for the type `TValue`: for example, 0 (zero) for integer types, `false` for Boolean types, and `null` for reference types.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyList<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Represents a read-only collection of elements that can be accessed by index.

C#

```
public interface IReadOnlyList<out T> : System.Collections.Generic.IEnumerable<out T>, System.Collections.Generic.IReadOnlyCollection<out T>
```

Type Parameters

T

The type of elements in the read-only list.

This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived. For more information about covariance and contravariance, see [Covariance and Contravariance in Generics](#).

Derived [Microsoft.Extensions.AI.GeneratedEmbeddings<TEmbedding>](#)

[Microsoft.Extensions.Primitives.StringValues](#)

[System.ArraySegment<T>](#)

[System.Collections.Generic.List<T>](#)

[System.Collections.Generic.OrderedDictionary< TKey, TValue >](#)

[More...](#)

Implements [IEnumerable<T>](#), [IReadOnlyCollection<T>](#), [IEnumerable](#)

Remarks

The [IReadOnlyList<T>](#) represents a list in which the number and order of list elements is read-only. The content of list elements is not guaranteed to be read-only.

Properties

 Expand table

Count	Gets the number of elements in the collection. (Inherited from IReadOnlyCollection<T>)
Item[Int32]	Gets the element at the specified index in the read-only list.

Methods

 [Expand table](#)

GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)
---------------------------------	--

Extension Methods

 [Expand table](#)

ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.

<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects,

	given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .

<code>Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Concatenates two sequences.
<code>Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)</code>	Determines whether a sequence contains a specified element by using a specified <code>IEqualityComparer<T></code> .
<code>Contains<TSource>(IEnumerable<TSource>, TSource)</code>	Determines whether a sequence contains a specified element by using the default equality comparer.
<code>Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns a number that represents how many elements in the specified sequence satisfy a condition.
<code>Count<TSource>(IEnumerable<TSource>)</code>	Returns the number of elements in a sequence.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>)</code>	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.

<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition

	<p>or a default value if no such element is found.</p>
FirstOrDefault<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence, or a default value if the sequence contains no elements.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Groups the elements of a sequence according to a specified key selector function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are

	projected by using a specified function.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on key equality and groups the results. A specified IEqualityComparer<T> is used to compare keys.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set intersection of two sequences by using the specified IEqualityComparer<T> to compare values.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set intersection of two sequences by using the default equality comparer to compare values.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set intersection of two sequences according to a specified key selector function.
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on matching keys. A specified IEqualityComparer<T> is used to compare keys.
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.

<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IQueryable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IQueryable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.

<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.

<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an <code>IEnumerable</code> based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.

<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
<code>Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
<code>Single<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.

SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's

	<p>index is used in the logic of the predicate function.</p>
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the sum of the sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.

	transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.
Take<TSource>(IEnumerable<TSource>, Range)	Returns a specified range of contiguous elements from a sequence.
TakeLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns elements from a sequence as long as a specified condition is true.
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource, TKey, TElement>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource, TElement>, IEqualityComparer<TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T>

<code>Comparer<TKey>()</code>	according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToList<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer and an element selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Lookup<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>TryGetNonEnumeratedCount<TSource>(IQueryable<TSource>, Int32)</code>	Attempts to determine the number of elements in a sequence without forcing an enumeration.
<code>Union<TSource>(IQueryable<TSource>, IQueryable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set union of two sequences by using a specified <code>IEqualityComparer<T></code> .
<code>Union<TSource>(IQueryable<TSource>, IQueryable<TSource>)</code>	Produces the set union of two sequences by using the default

	equality comparer.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>Where<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Filters a sequence of values based on a predicate.
<code>Where<TSource>(IQueryable<TSource>, Func<TSource,Int32,Boolean>)</code>	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
<code>Zip<TFirst,TSecond>(IQueryable<TFirst>, IQueryable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IQueryable<TFirst>, IQueryable<TSecond>, IQueryable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IQueryable<TFirst>, IQueryable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IQueryable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IQueryable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IQueryable)</code>	Converts an <code>IEnumerable</code> to an <code>IQueryable</code> .
<code>AsQueryable<TElement>(IQueryable<TElement>)</code>	Converts a generic <code>IEnumerable<T></code> to a generic <code>IQueryable<T></code> .
<code>Ancestors<T>(IQueryable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching <code>XName</code> are included in the collection.
<code>Ancestors<T>(IQueryable<T>)</code>	Returns a collection of elements that contains the ancestors of

	every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlyList<T>.Item[Int32] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the element at the specified index in the read-only list.

C#

```
public T this[int index] { get; }
```

Parameters

index [Int32](#)

The zero-based index of the element to get.

Property Value

T

The element at the specified index in the read-only list.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

IReadOnlySet<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Provides a readonly abstraction of a set.

C#

```
public interface IReadOnlySet<T> : System.Collections.Generic.IEnumerable<T>,
System.Collections.Generic.IReadOnlyCollection<T>
```

Type Parameters

T

The type of elements in the set.

Derived [System.Collections.Generic.HashSet<T>](#)

[System.Collections.Generic.SortedSet<T>](#)

[System.Collections.Immutable.ImmutableHashSet<T>](#)

[System.Collections.Immutable.ImmutableSortedSet<T>](#)

[More...](#)

Implements [IEnumerable<T>](#) , [IReadOnlyCollection<T>](#) , [IEnumerable](#)

Properties

[+] [Expand table](#)

Count	Gets the number of elements in the collection. (Inherited from IReadOnlyCollection<T>)
-------	--

Methods

[+] [Expand table](#)

Contains(T)	Determines if the set contains a specific item.
-----------------------------	---

GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)
IsProperSubset Of(IEnumerable<T>)	Determines whether the current set is a proper (strict) subset of a specified collection.
IsProperSuperset Of(IEnumerable<T>)	Determines whether the current set is a proper (strict) superset of a specified collection.
IsSubsetOf(IEnumerable<T>)	Determine whether the current set is a subset of a specified collection.
IsSupersetOf(IEnumerable<T>)	Determine whether the current set is a super set of a specified collection.
Overlaps(IEnumerable<T>)	Determines whether the current set overlaps with the specified collection.
SetEquals(IEnumerable<T>)	Determines whether the current set and the specified collection contain the same elements.

Extension Methods

 [Expand table](#)

ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its

	contents by using the specified key comparer.
ToImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
ToImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.
ToImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
ToImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
ToImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
ToImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .

<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform

	function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the average of a sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform

	function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most <code>size</code> .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.

<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default

	<p>value if the sequence contains no elements.</p>
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	<p>Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.</p>
<code>FirstOrDefault<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	<p>Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.</p>
<code>FirstOrDefault<TSource>(IQueryable<TSource>)</code>	<p>Returns the first element of a sequence, or a default value if the sequence contains no elements.</p>
<code>GroupBy<TSource,TKey>(IQueryable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.</p>
<code>GroupBy<TSource,TKey>(IQueryable<TSource>, Func<TSource,TKey>)</code>	<p>Groups the elements of a sequence according to a specified key selector function.</p>
<code>GroupBy<TSource,TKey,TElement>(IQueryable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	<p>Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.</p>
<code>GroupBy<TSource,TKey,TElement>(IQueryable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.</p>
<code>GroupBy<TSource,TKey,TResult>(IQueryable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.</p>
<code>GroupBy<TSource,TKey,TResult>(IQueryable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a</p>

	<p>result value from each group and its key.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	<p>Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	<p>Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	<p>Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	<p>Produces the set intersection of two sequences by using the default equality comparer to compare values.</p>
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	<p>Produces the set intersection of two sequences according to a specified key selector function.</p>
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	<p>Produces the set intersection of two sequences according to a specified key selector function.</p>

<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum Decimal value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the maximum Double value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the maximum Single value.
Max<TSource>(IEnumerable<TSource>)	Returns the maximum value in a generic sequence.
Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Invokes a transform function on each element of a generic

	sequence and returns the maximum resulting value.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function.
Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and

	returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in descending order according to a key.
Prepend<TSource>(IEnumerable<TSource>, TSource)	Adds a value to the beginning of the sequence.

<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified <code>IEqualityComparer<T></code> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.

Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.

<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the sum of the sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key

	selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to specified key selector and element selector functions.

TryGetNonEnumeratedCount<TSource>(IQueryable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IQueryable<TSource>, IQueryable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IQueryable<TSource>, IQueryable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IQueryable<TSource>, IQueryable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IQueryable<TSource>, IQueryable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IQueryable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IQueryable<TFirst>, IQueryable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IQueryable<TFirst>, IQueryable<TSecond>, IQueryable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IQueryable<TFirst>, IQueryable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
AsParallel(IQueryable)	Enables parallelization of a query.
AsParallel<TSource>(IQueryable<TSource>)	Enables parallelization of a query.
AsQueryable(IQueryable)	Converts an IQueryable to an IQueryable .
AsQueryable<TElement>(IQueryable<TElement>)	Converts a generic IQueryable<T> to a generic IQueryable<T> .

Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.

[Remove<T>\(IEnumerable<T>\)](#)

Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

IReadOnlySet<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Determines if the set contains a specific item.

C#

```
public bool Contains(T item);
```

Parameters

item T

The item to check if the set contains.

Returns

[Boolean](#)

`true` if found; otherwise `false`.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

IReadOnlySet<T>.IsProperSubsetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Determines whether the current set is a proper (strict) subset of a specified collection.

C#

```
public bool IsProperSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is a proper subset of other; otherwise `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

IReadOnlySet<T>.IsProperSupersetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Determines whether the current set is a proper (strict) superset of a specified collection.

C#

```
public bool IsProperSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the collection is a proper superset of other; otherwise `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

IReadOnlySet<T>.IsSubsetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Determine whether the current set is a subset of a specified collection.

C#

```
public bool IsSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is a subset of `other`; otherwise `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

IReadOnlySet<T>.IsSupersetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Determine whether the current set is a super set of a specified collection.

C#

```
public bool IsSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is a super set of other; otherwise `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

IReadOnly Set<T>.Overlaps(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Determines whether the current set overlaps with the specified collection.

C#

```
public bool Overlaps(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set and other share at least one common element; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

IReadOnlySet<T>.SetEquals(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Runtime.dll

Determines whether the current set and the specified collection contain the same elements.

C#

```
public bool SetEquals(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is equal to other; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

ISet<T> Interface

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Provides the base interface for the abstraction of sets.

C#

```
public interface ISet<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>
```

Type Parameters

T

The type of elements in the set.

Derived [System.Collections.Frozen.FrozenSet<T>](#)

[System.Collections.Generic.HashSet<T>](#)

[System.Collections.Generic.SortedSet<T>](#)

[System.Collections.Immutable.ImmutableHashSet<T>](#)

[System.Collections.Immutable.ImmutableHashSet<T>.Builder](#)

More...

Implements [ICollection<T>](#) , [IEnumerable<T>](#) , [IQueryable](#)

Remarks

This interface provides methods for implementing sets, which are collections that have unique elements and specific operations. The [HashSet<T>](#) and [SortedSet<T>](#) collections implement this interface.

Properties

 Expand table

Count	Gets the number of elements contained in the ICollection<T> . (Inherited from ICollection<T>)
-------	---

IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only. (Inherited from ICollection<T>)
----------------------------	--

Methods

[\[+\] Expand table](#)

Add(T)	Adds an element to the current set and returns a value to indicate if the element was successfully added.
Clear()	Removes all items from the ICollection<T> . (Inherited from ICollection<T>)
Contains(T)	Determines whether the ICollection<T> contains a specific value. (Inherited from ICollection<T>)
CopyTo(T[], Int32)	Copies the elements of the ICollection<T> to an Array , starting at a particular Array index. (Inherited from ICollection<T>)
ExceptWith(IEnumerable<T>)	Removes all elements in the specified collection from the current set.
GetEnumerator()	Returns an enumerator that iterates through a collection. (Inherited from IEnumerable)
IntersectWith(IEnumerable<T>)	Modifies the current set so that it contains only elements that are also in a specified collection.
IsProperSubset Of(IEnumerable<T>)	Determines whether the current set is a proper (strict) subset of a specified collection.
IsProperSuperset Of(IEnumerable<T>)	Determines whether the current set is a proper (strict) superset of a specified collection.
IsSubsetOf(IEnumerable<T>)	Determines whether a set is a subset of a specified collection.
IsSupersetOf(IEnumerable<T>)	Determines whether the current set is a superset of a specified collection.
Overlaps(IEnumerable<T>)	Determines whether the current set overlaps with the specified collection.
Remove(T)	Removes the first occurrence of a specific object from the ICollection<T> . (Inherited from ICollection<T>)
SetEquals(IEnumerable<T>)	Determines whether the current set and the specified collection contain the same elements.

<code>SymmetricExceptWith(IEnumerable<T>)</code>	Modifies the current set so that it contains only elements that are present either in the current set or in the specified collection, but not both.
<code>UnionWith(IEnumerable<T>)</code>	Modifies the current set so that it contains all elements that are present in the current set, in the specified collection, or in both.

Extension Methods

[Expand table](#)

<code>ToImmutableArray<TSource>(IEnumerable<TSource>)</code>	Creates an immutable array from the specified collection.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of

	its contents.
ToImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter T is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter T is DataRow .
CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T> object where the generic parameter T is DataRow .
Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)	Applies an accumulator function over a sequence.

<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most <code>size</code> .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .

Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEqualityComparer<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEqualityComparer<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IList<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IReadOnlyList<TSource>, Int32)	Returns the element at a specified index in a sequence.

ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set difference of two sequences according to a specified key selector function.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set difference of two sequences according to a specified key selector function.
First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element in a sequence that satisfies a specified condition.
First<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence.
FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence, or a default value if the sequence contains no elements.

<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a

	<p>result value from each group and its key. The elements of each group are projected by using a specified function.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.

<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Decimal>)</code>	Invokes a transform function on each element of a sequence and

	returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Invokes a transform function on each element of a generic

	sequence and returns the minimum resulting value.
MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Sorts the elements of a sequence in descending order according to a key.
Prepend<TSource>(IEnumerable<TSource>, TSource)	Adds a value to the beginning of the sequence.
Reverse<TSource>(IEnumerable<TSource>)	Inverts the order of the elements in a sequence.
Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Projects each element of a sequence into a new form.
Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)	Projects each element of a sequence into a new form by incorporating the element's index.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> ,

	<p>and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.</p>
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.

<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>Skip<TSource>(IEnumerable<TSource>, Int32)</code>	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
<code>SkipLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Computes the sum of the sequence of nullable <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Computes the sum of the sequence of <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.

<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToListAsync<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer and an element selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)</code>	Attempts to determine the number of elements in a sequence without forcing an enumeration.
<code>Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set union of two sequences by using a specified <code>IEqualityComparer<T></code> .
<code>Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set union of two sequences by using the default equality comparer.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.

Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the

	descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Adds an element to the current set and returns a value to indicate if the element was successfully added.

C#

```
public bool Add(T item);
```

Parameters

item **T**

The element to add to the set.

Returns

Boolean

`true` if the element is added to the set; `false` if the element is already in the set.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.ExceptWith(IEnumerable<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Removes all elements in the specified collection from the current set.

C#

```
public void ExceptWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection of items to remove from the set.

Exceptions

[ArgumentNullException](#)

other is `null`.

Remarks

This method is an $O(n)$ operation, where n is the number of elements in the **other** parameter.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.IntersectWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Modifies the current set so that it contains only elements that are also in a specified collection.

C#

```
public void IntersectWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Exceptions

[ArgumentNullException](#)

other is `null`.

Remarks

This method ignores any duplicate elements in **other**.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.IsProperSubset Of(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the current set is a proper (strict) subset of a specified collection.

C#

```
public bool IsProperSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is a proper subset of `other`; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

If the current set is a proper subset of `other`, `other` must have at least one element that the current set does not have.

An empty set is a proper subset of any other collection. Therefore, this method returns `true` if the current set is empty, unless the `other` parameter is also an empty set.

This method always returns `false` if the current set has more or the same number of elements than `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.IsProperSuperset Of(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the current set is a proper (strict) superset of a specified collection.

C#

```
public bool IsProperSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is a proper superset of `other`; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

If the current set is a proper superset of `other`, the current set must have at least one element that `other` does not have.

An empty set is a proper superset of any other collection. Therefore, this method returns `true` if the collection represented by the `other` parameter is empty, unless the current set is also empty.

This method always returns `false` if the number of elements in the current set is less than or equal to the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.IsSubsetOf(IEnumerable<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether a set is a subset of a specified collection.

C#

```
public bool IsSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is a subset of `other`; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

If `other` contains the same elements as the current set, the current set is still considered a subset of `other`.

This method always returns `false` if the current set has elements that are not in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.IsSupersetOf(IEnumerable<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the current set is a superset of a specified collection.

C#

```
public bool IsSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is a superset of `other`; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

If `other` contains the same elements as the current set, the current set is still considered a superset of `other`.

This method always returns `false` if the current set has fewer elements than `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.Overlaps(IEnumerable<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the current set overlaps with the specified collection.

C#

```
public bool Overlaps(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set and `other` share at least one common element; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

Any duplicate elements in `other` are ignored.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.SetEquals(IEnumerable<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Determines whether the current set and the specified collection contain the same elements.

C#

```
public bool SetEquals(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Returns

[Boolean](#)

`true` if the current set is equal to `other`; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

This method ignores the order of elements and any duplicate elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.SymmetricExceptWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Modifies the current set so that it contains only elements that are present either in the current set or in the specified collection, but not both.

C#

```
public void SymmetricExceptWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

Any duplicate elements in `other` are ignored.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

ISet<T>.UnionWith(IEnumerable<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Modifies the current set so that it contains all elements that are present in the current set, in the specified collection, or in both.

C#

```
public void UnionWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current set.

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

Any duplicate elements in `other` are ignored.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

KeyNotFoundException Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

The exception that is thrown when the key specified for accessing an element in a collection does not match any key in the collection.

C#

```
public class KeyNotFoundException : SystemException
```

Inheritance [Object](#) → [Exception](#) → [SystemException](#) → [KeyNotFoundException](#)

Remarks

A [KeyNotFoundException](#) is thrown when an operation attempts to retrieve an element from a collection using a key that does not exist in that collection.

[KeyNotFoundException](#) uses the HRESULT COR_E_KEYNOTFOUND, which has the value 0x80131577.

For a list of initial property values for an instance of the [KeyNotFoundException](#) class, see the [KeyNotFoundException constructors](#).

Constructors

[+] Expand table

KeyNotFoundException()	Initializes a new instance of the KeyNotFoundException class using default property values.
KeyNotFoundException(SerializationInfo, StreamingContext)	Initializes a new instance of the KeyNotFoundException class with serialized data.
KeyNotFoundException(String, Exception)	Initializes a new instance of the KeyNotFoundException class with the specified error message and a reference to the inner exception that is the cause of this exception.

KeyNotFoundException(String)	Initializes a new instance of the KeyNotFoundException class with the specified error message.
--	--

Properties

[+] [Expand table](#)

Data	Gets a collection of key/value pairs that provide additional user-defined information about the exception. (Inherited from Exception)
HelpLink	Gets or sets a link to the help file associated with this exception. (Inherited from Exception)
HResult	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception. (Inherited from Exception)
InnerException	Gets the Exception instance that caused the current exception. (Inherited from Exception)
Message	Gets a message that describes the current exception. (Inherited from Exception)
Source	Gets or sets the name of the application or the object that causes the error. (Inherited from Exception)
StackTrace	Gets a string representation of the immediate frames on the call stack. (Inherited from Exception)
TargetSite	Gets the method that throws the current exception. (Inherited from Exception)

Methods

[+] [Expand table](#)

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetBaseException()	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions. (Inherited from Exception)
GetHashCode()	Serves as the default hash function. (Inherited from Object)

GetObjectData(SerializationInfo, StreamingContext)	When overridden in a derived class, sets the SerializationInfo with information about the exception. (Inherited from Exception)
GetType()	Gets the runtime type of the current instance. (Inherited from Exception)
MemberwiseClone()	Creates a shallow copy of the current Object . (Inherited from Object)
ToString()	Creates and returns a string representation of the current exception. (Inherited from Exception)

Events

 [Expand table](#)

SerializeObject	Obsolete.
State	Occurs when an exception is serialized to create an exception state object that contains serialized data about the exception. (Inherited from Exception)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

KeyNotFoundException Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Initializes a new instance of the [KeyNotFoundException](#) class.

Overloads

[+] [Expand table](#)

KeyNotFoundException()	Initializes a new instance of the KeyNotFoundException class using default property values.
KeyNotFoundException(String)	Initializes a new instance of the KeyNotFoundException class with the specified error message.
KeyNotFoundException(SerializationInfo, StreamingContext)	Initializes a new instance of the KeyNotFoundException class with serialized data.
KeyNotFoundException(String, Exception)	Initializes a new instance of the KeyNotFoundException class with the specified error message and a reference to the inner exception that is the cause of this exception.

KeyNotFoundException()

Initializes a new instance of the [KeyNotFoundException](#) class using default property values.

C#

```
public KeyNotFoundException();
```

Remarks

This constructor initializes the [Message](#) property of the new instance to a system-supplied message that describes the error, such as "The given key was not present in the dictionary." This message takes into account the current system culture.

The following table shows the initial property values for an instance of the [KeyNotFoundException](#) class.

[+] Expand table

Property	Value
InnerException	<code>null</code> .
Message	The localized error message string.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

KeyNotFoundException(String)

Initializes a new instance of the [KeyNotFoundException](#) class with the specified error message.

C#

```
public KeyNotFoundException(string? message);
```

Parameters

message `String`

The message that describes the error.

Remarks

The content of the `message` parameter is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current

system culture.

The following table shows the initial property values for an instance of the [KeyNotFoundException](#) class.

 [Expand table](#)

Property	Value
InnerException	<code>null</code> .
Message	The localized error message string.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

[KeyNotFoundException\(SerializationInfo, StreamingContext\)](#)

Initializes a new instance of the [KeyNotFoundException](#) class with serialized data.

C#

```
protected KeyNotFoundException(System.Runtime.Serialization.SerializationInfo
info, System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

The [SerializationInfo](#) that holds the serialized object data about the exception being thrown.

context [StreamingContext](#)

The [StreamingContext](#) that contains contextual information about the source or destination.

Remarks

This constructor is called during deserialization to reconstitute the exception object transmitted over a stream. For more information, see [XML and SOAP Serialization](#).

Applies to

- ▼ .NET 10 and other versions

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

KeyNotFoundException(String, Exception)

Initializes a new instance of the [KeyNotFoundException](#) class with the specified error message and a reference to the inner exception that is the cause of this exception.

C#

```
public KeyNotFoundException(string? message, Exception? innerException);
```

Parameters

message [String](#)

The error message that explains the reason for the exception.

innerException [Exception](#)

The exception that is the cause of the current exception. If the `innerException` parameter is not `null`, the current exception is raised in a `catch` block that handles the inner exception.

Remarks

An exception that is thrown as a direct result of a previous exception should include a reference to the previous exception in the [InnerException](#) property. The [InnerException](#)

property returns the same value that is passed into the constructor, or `null` if the [InnerException](#) property does not supply the inner exception value to the constructor.

The following table shows the initial property values for an instance of the [KeyNotFoundException](#) class.

 Expand table

Property	Value
InnerException	The inner exception reference.
Message	The localized error message string.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

KeyValuePair Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Creates instances of the [KeyValuePair<TKey,TValue>](#) struct.

C#

```
public static class KeyValuePair
```

Inheritance [Object](#) → KeyValuePair

Methods

 [Expand table](#)

Create<TKey,TValue>(TKey, TValue)	Creates a new key/value pair instance using provided values.
---	--

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

KeyValuePair.Create< TKey, TValue >(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Creates a new key/value pair instance using provided values.

C#

```
public static System.Collections.Generic.KeyValuePair< TKey, TValue >
Create< TKey, TValue >( TKey key, TValue value );
```

Type Parameters

TKey

The type of the key.

TValue

The type of the value.

Parameters

key TKey

The key of the new [KeyValuePair< TKey, TValue >](#) to be created.

value TValue

The value of the new [KeyValuePair< TKey, TValue >](#) to be created.

Returns

[KeyValuePair< TKey, TValue >](#)

A key/value pair containing the provided arguments as values.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

KeyValuePair<TKey,TValue> Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Defines a key/value pair that can be set or retrieved.

C#

```
public readonly struct KeyValuePair<TKey, TValue>
```

Type Parameters

TKey

The type of the key.

TValue

The type of the value.

Inheritance [Object](#) → [ValueType](#) → KeyValuePair<TKey,TValue>

Examples

The following code example shows how to enumerate the keys and values in a dictionary, using the [KeyValuePair<TKey,TValue>](#) structure.

This code is part of a larger example provided for the [Dictionary<TKey,TValue>](#) class.

C#

```
// 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);
}
```

Remarks

The `Dictionary<TKey,TValue>.Enumerator.Current` property returns an instance of this type.

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 a collection based on `IDictionary<TKey,TValue>` 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 `KeyValuePair<TKey,TValue>`. For example:

C#

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

The `foreach` statement is a wrapper around the enumerator, which allows only reading from, not writing to, the collection.

Constructors

[+] Expand table

<code>KeyValuePair<TKey,TValue>(TKey, TValue)</code>	Initializes a new instance of the <code>KeyValuePair<TKey,TValue></code> structure with the specified key and value.
--	--

Properties

[+] Expand table

<code>Key</code>	Gets the key in the key/value pair.
<code>Value</code>	Gets the value in the key/value pair.

Methods

[+] Expand table

<code>Deconstruct(TKey, TValue)</code>	Deconstructs the current <code>KeyValuePair<TKey,TValue></code> .
--	---

ToString()	Returns a string representation of the KeyValuePair<TKey,TValue> , using the string representations of the key and value.
----------------------------	---

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionary<TKey,TValue>](#)
- [Dictionary<TKey,TValue>.Enumerator](#)
- [DictionaryEntry](#)

KeyValuePair<TKey,TValue>(TKey, TValue) Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Initializes a new instance of the `KeyValuePair<TKey,TValue>` structure with the specified key and value.

C#

```
public KeyValuePair(TKey key, TValue value);
```

Parameters

key TKey

The object defined in each key/value pair.

value TValue

The definition associated with `key`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

KeyValuePair<TKey,TValue>.Key Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the key in the key/value pair.

C#

```
public TKey Key { get; }
```

Property Value

TKey

A `TKey` that is the key of the [KeyValuePair<TKey,TValue>](#).

Examples

The following code example shows how to enumerate the keys and values in a dictionary, using the [KeyValuePair<TKey,TValue>](#) structure.

This code is part of a larger example provided for the [Dictionary<TKey,TValue>](#) class.

C#

```
// 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);
}
```

Remarks

This property is read-only.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

KeyValuePair< TKey, TValue >.Value Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Gets the value in the key/value pair.

C#

```
public TValue Value { get; }
```

Property Value

TValue

A `TValue` that is the value of the [KeyValuePair< TKey, TValue >](#).

Examples

The following code example shows how to enumerate the keys and values in a dictionary, using the [KeyValuePair< TKey, TValue >](#) structure.

This code is part of a larger example provided for the [Dictionary< TKey, TValue >](#) class.

C#

```
// 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);
}
```

Remarks

This property is read-only.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

KeyValuePair<TKey,TValue>.Deconstruct(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Deconstructs the current [KeyValuePair<TKey,TValue>](#).

C#

```
public void Deconstruct(out TKey key, out TValue value);
```

Parameters

key TKey

The key of the current [KeyValuePair<TKey,TValue>](#).

value TValue

The value of the current [KeyValuePair<TKey,TValue>](#).

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

KeyValuePair< TKey, TValue >.ToString Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Runtime.dll

Returns a string representation of the [KeyValuePair< TKey, TValue >](#), using the string representations of the key and value.

C#

```
public override string ToString();
```

Returns

[String](#)

A string representation of the [KeyValuePair< TKey, TValue >](#), which includes the string representations of the key and value.

Remarks

The string representation consists of the string representations of the key and value, separated by a comma and a space, and enclosed in square brackets. For example, the [ToString](#) method for a [KeyValuePair< TKey, TValue >](#) structure with the string [Key](#) "Test" and the integer [Value](#) 14 returns the string "[Test, 14]".

! Note

This method uses the [ToString](#) methods provided by the key and value types. Some types do not return useful or informative values for their [ToString](#) methods.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [LinkedList<T>](#).

C#

```
public struct LinkedList<T>.Enumerator :  
    System.Collections.Generic.IEnumerator<T>,  
    System.Runtime.Serialization.IDeserializationCallback,  
    System.Runtime.Serialization.ISerializable
```

Type Parameters

T

Inheritance [Object](#) → [ValueType](#) → [LinkedList<T>.Enumerator](#)

Implements [IEnumerator<T>](#) , [IEnumerator](#) , [IDisposable](#) , [IDeserializationCallback](#) , [ISerializable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Properties

[+] Expand table

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

[+] Expand table

Dispose()	Releases all resources used by the LinkedList<T>.Enumerator .
MoveNext()	Advances the enumerator to the next element of the LinkedList<T> .

Explicit Interface Implementations

[+] Expand table

IDeserializationCallback.OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
--	---

IEnumerator.Current	Gets the element at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection. This class cannot be inherited.
ISerializable.GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data needed to serialize the LinkedList<T> instance.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

LinkedList<T>.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the [LinkedList<T>](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

LinkedList<T>.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [LinkedList<T>.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [LinkedList<T>](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to [MoveNext](#) advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

LinkedList<T>.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

`IEnumerator.Current` does not move the position of the enumerator, and consecutive calls to `IEnumerator.Current` return the same object until either `MoveNext` or `IEnumerator.Reset` is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator.Reset\(\)](#)
- [MoveNext\(\)](#)

LinkedList<T>.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.
This class cannot be inherited.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [Current](#)
- [IEnumerator.Current](#)
- [MoveNext\(\)](#)

LinkedList<T>.Enumerator.IDeserializationCallback.OnDeserialization Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and raises the deserialization event when the deserialization is complete.

C#

```
void IDeserializationCallback.OnDeserialization(object sender);
```

Parameters

sender [Object](#)

The source of the deserialization event.

Implements

[OnDeserialization\(Object\)](#)

Exceptions

[SerializationException](#)

The [SerializationInfo](#) object associated with the current [LinkedList<T>](#) instance is invalid.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

See also

- [ISerializable](#)
- [SerializationInfo](#)
- [StreamingContext](#)
- [GetObjectData\(SerializationInfo, StreamingContext\)](#)

LinkedList<T>.Enumerator.ISerializable.GetObjectData Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and returns the data needed to serialize the [LinkedList<T>](#) instance.

C#

```
void ISerializable.GetObjectData(System.Runtime.Serialization.SerializationInfo  
info, System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information required to serialize the [LinkedList<T>](#) instance.

context [StreamingContext](#)

A [StreamingContext](#) object that contains the source and destination of the serialized stream associated with the [LinkedList<T>](#) instance.

Implements

[GetObjectData\(SerializationInfo, StreamingContext\)](#)

Exceptions

[ArgumentNullException](#)

`info` is `null`.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

See also

- [ISerializable](#)
- [SerializationInfo](#)
- [StreamingContext](#)
- [OnDeserialization\(Object\)](#)

LinkedList<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a doubly linked list.

C#

```
public class LinkedList<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>,
System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.ICollection,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

Type Parameters

T

Specifies the element type of the linked list.

Inheritance [Object](#) → [LinkedList<T>](#)

Implements [ICollection<T>](#) , [IEnumerable<T>](#) , [IReadOnlyCollection<T>](#) , [ICollection](#) ,
[IEnumerable](#) , [IDeserializationCallback](#) , [ISerializable](#)

Examples

The following code example demonstrates many features of the [LinkedList<T>](#) class.

C#

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

public class Example
{
    public static void Main()
    {
        // Create the link list.
        string[] words =
            { "the", "fox", "jumps", "over", "the", "dog" };
    }
}
```

```

LinkedList<string> sentence = new LinkedList<string>(words);
Display(sentence, "The linked list values:");

// Add the word 'today' to the beginning of the linked list.
sentence.AddFirst("today");
Display(sentence, "Test 1: Add 'today' to beginning of the list:");

// Move the first node to be the last node.
LinkedListNode<string> mark1 = sentence.First;
sentence.RemoveFirst();
sentence.AddLast(mark1);
Display(sentence, "Test 2: Move first node to be last node:");

// Change the last node to 'yesterday'.
sentence.RemoveLast();
sentence.AddLast("yesterday");
Display(sentence, "Test 3: Change the last node to 'yesterday':");

// Move the last node to be the first node.
mark1 = sentence.Last;
sentence.RemoveLast();
sentence.AddFirst(mark1);
Display(sentence, "Test 4: Move last node to be first node:");

// Indicate the last occurrence of 'the'.
sentence.RemoveFirst();
LinkedListNode<string> current = sentence.FindLast("the");
IndicateNode(current, "Test 5: Indicate last occurrence of 'the':");

// Add 'lazy' and 'old' after 'the' (the ListNode named current).
sentence.AddAfter(current, "old");
sentence.AddAfter(current, "lazy");
IndicateNode(current, "Test 6: Add 'lazy' and 'old' after 'the':");

// Indicate 'fox' node.
current = sentence.Find("fox");
IndicateNode(current, "Test 7: Indicate the 'fox' node:");

// Add 'quick' and 'brown' before 'fox':
sentence.AddBefore(current, "quick");
sentence.AddBefore(current, "brown");
IndicateNode(current, "Test 8: Add 'quick' and 'brown' before 'fox':");

// Keep a reference to the current node, 'fox',
// and to the previous node in the list. Indicate the 'dog' node.
mark1 = current;
LinkedListNode<string> mark2 = current.Previous;
current = sentence.Find("dog");
IndicateNode(current, "Test 9: Indicate the 'dog' node:");

// The AddBefore method throws an InvalidOperationException
// if you try to add a node that already belongs to a list.
Console.WriteLine("Test 10: Throw exception by adding node (fox) already
in the list:");
try

```

```

{
    sentence.AddBefore(current, mark1);
}
catch (InvalidOperationException ex)
{
    Console.WriteLine("Exception message: {0}", ex.Message);
}
Console.WriteLine();

// Remove the node referred to by mark1, and then add it
// before the node referred to by current.
// Indicate the node referred to by current.
sentence.Remove(mark1);
sentence.AddBefore(current, mark1);
IndicateNode(current, "Test 11: Move a referenced node (fox) before the
current node (dog):");

// Remove the node referred to by current.
sentence.Remove(current);
IndicateNode(current, "Test 12: Remove current node (dog) and attempt to
indicate it:");

// Add the node after the node referred to by mark2.
sentence.AddAfter(mark2, current);
IndicateNode(current, "Test 13: Add node removed in test 12 after a
referenced node (brown):");

// The Remove method finds and removes the
// first node that has the specified value.
sentence.Remove("old");
Display(sentence, "Test 14: Remove node that has the value 'old':");

// When the linked list is cast to ICollection(Of String),
// the Add method adds a node to the end of the list.
sentence.RemoveLast();
ICollection<string> icoll = sentence;
icoll.Add("rhinoceros");
Display(sentence, "Test 15: Remove last node, cast to ICollection, and add
'rhinoceros':");

Console.WriteLine("Test 16: Copy the list to an array:");
// Create an array with the same number of
// elements as the linked list.
string[] sArray = new string[sentence.Count];
sentence.CopyTo(sArray, 0);

foreach (string s in sArray)
{
    Console.WriteLine(s);
}

Console.WriteLine("Test 17: linked list Contains 'jumps' = {0}",
    sentence.Contains("jumps"));

// Release all the nodes.

```

```

sentence.Clear();

Console.WriteLine();
Console.WriteLine("Test 18: Cleared linked list Contains 'jumps' = {0}",
    sentence.Contains("jumps"));

Console.ReadLine();
}

private static void Display(LinkedList<string> words, string test)
{
    Console.WriteLine(test);
    foreach (string word in words)
    {
        Console.Write(word + " ");
    }
    Console.WriteLine();
    Console.WriteLine();
}

private static void IndicateNode(LinkedListNode<string> node, string test)
{
    Console.WriteLine(test);
    if (node.List == null)
    {
        Console.WriteLine("Node '{0}' is not in the list.\n",
            node.Value);
        return;
    }

    StringBuilder result = new StringBuilder("(" + node.Value + ")");
    LinkedListNode<string> nodeP = node.Previous;

    while (nodeP != null)
    {
        result.Insert(0, nodeP.Value + " ");
        nodeP = nodeP.Previous;
    }

    node = node.Next;
    while (node != null)
    {
        result.Append(" " + node.Value);
        node = node.Next;
    }

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

//This code example produces the following output:
//
//The linked list values:
//the fox jumps over the dog

```

```
//Test 1: Add 'today' to beginning of the list:  
//today the fox jumps over the dog  
  
//Test 2: Move first node to be last node:  
//the fox jumps over the dog today  
  
//Test 3: Change the last node to 'yesterday':  
//the fox jumps over the dog yesterday  
  
//Test 4: Move last node to be first node:  
//yesterday the fox jumps over the dog  
  
//Test 5: Indicate last occurrence of 'the':  
//the fox jumps over (the) dog  
  
//Test 6: Add 'lazy' and 'old' after 'the':  
//the fox jumps over (the) lazy old dog  
  
//Test 7: Indicate the 'fox' node:  
//the (fox) jumps over the lazy old dog  
  
//Test 8: Add 'quick' and 'brown' before 'fox':  
//the quick brown (fox) jumps over the lazy old dog  
  
//Test 9: Indicate the 'dog' node:  
//the quick brown fox jumps over the lazy old (dog)  
  
//Test 10: Throw exception by adding node (fox) already in the list:  
//Exception message: The LinkedList node belongs a LinkedList.  
  
//Test 11: Move a referenced node (fox) before the current node (dog):  
//the quick brown jumps over the lazy old fox (dog)  
  
//Test 12: Remove current node (dog) and attempt to indicate it:  
//Node 'dog' is not in the list.  
  
//Test 13: Add node removed in test 12 after a referenced node (brown):  
//the quick brown (dog) jumps over the lazy old fox  
  
//Test 14: Remove node that has the value 'old':  
//the quick brown dog jumps over the lazy fox  
  
//Test 15: Remove last node, cast to ICollection, and add 'rhinoceros':  
//the quick brown dog jumps over the lazy rhinoceros  
  
//Test 16: Copy the list to an array:  
//the  
//quick  
//brown  
//dog  
//jumps  
//over  
//the  
//lazy
```

```
//rhinoceros

//Test 17: linked list Contains 'jumps'= True

//Test 18: Cleared linked list Contains 'jumps' = False
//
```

Remarks

[LinkedList<T>](#) is a general-purpose linked list. It supports enumerators and implements the [ICollection](#) interface, consistent with other collection classes in the .NET Framework.

[LinkedList<T>](#) provides separate nodes of type [LinkedListNode<T>](#), so insertion and removal are O(1) operations.

You can remove nodes and reinsert them, either in the same list or in another list, which results in no additional objects allocated on the heap. Because the list also maintains an internal count, getting the [Count](#) property is an O(1) operation.

Each node in a [LinkedList<T>](#) object is of the type [LinkedListNode<T>](#). Because the [LinkedList<T>](#) is doubly linked, each node points forward to the [Next](#) node and backward to the [Previous](#) node.

Lists that contain reference types perform better when a node and its value are created at the same time. [LinkedList<T>](#) accepts `null` as a valid [Value](#) property for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the [First](#) and [Last](#) properties contain `null`.

The [LinkedList<T>](#) class does not support chaining, splitting, cycles, or other features that can leave the list in an inconsistent state. The list remains consistent on a single thread. The only multithreaded scenario supported by [LinkedList<T>](#) is multithreaded read operations.

Constructors

 Expand table

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

<code>LinkedList<T>(SerializationInfo, StreamingContext)</code>	Initializes a new instance of the <code>LinkedList<T></code> class that is serializable with the specified <code>SerializationInfo</code> and <code>StreamingContext</code> .
---	---

Properties

[+] [Expand table](#)

<code>Count</code>	Gets the number of nodes actually contained in the <code>LinkedList<T></code> .
<code>First</code>	Gets the first node of the <code>LinkedList<T></code> .
<code>Last</code>	Gets the last node of the <code>LinkedList<T></code> .

Methods

[+] [Expand table](#)

<code>AddAfter(LinkedListNode<T>, LinkedListNode<T>)</code>	Adds the specified new node after the specified existing node in the <code>LinkedList<T></code> .
<code>AddAfter(LinkedListNode<T>, T)</code>	Adds a new node containing the specified value after the specified existing node in the <code>LinkedList<T></code> .
<code>AddBefore(LinkedListNode<T>, LinkedListNode<T>)</code>	Adds the specified new node before the specified existing node in the <code>LinkedList<T></code> .
<code>AddBefore(LinkedListNode<T>, T)</code>	Adds a new node containing the specified value before the specified existing node in the <code>LinkedList<T></code> .
<code>AddFirst(LinkedListNode<T>)</code>	Adds the specified new node at the start of the <code>LinkedList<T></code> .
<code>AddFirst(T)</code>	Adds a new node containing the specified value at the start of the <code>LinkedList<T></code> .
<code>AddLast(LinkedListNode<T>)</code>	Adds the specified new node at the end of the <code>LinkedList<T></code> .
<code>AddLast(T)</code>	Adds a new node containing the specified value at the end of the <code>LinkedList<T></code> .
<code>Clear()</code>	Removes all nodes from the <code>LinkedList<T></code> .
<code>Contains(T)</code>	Determines whether a value is in the <code>LinkedList<T></code> .
<code>CopyTo(T[], Int32)</code>	Copies the entire <code>LinkedList<T></code> to a compatible one-dimensional <code>Array</code> , 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)
Find(T)	Finds the first node that contains the specified value.
FindLast(T)	Finds the last node that contains the specified value.
GetEnumerator()	Returns an enumerator that iterates through the LinkedList<T> .
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data needed to serialize the LinkedList<T> instance.
GetType()	Gets the Type of the current instance. (Inherited from Object)
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(LinkedListNode<T>)	Removes the specified node from the LinkedList<T> .
Remove(T)	Removes the first occurrence of the specified value from the LinkedList<T> .
RemoveFirst()	Removes the node at the start of the LinkedList<T> .
RemoveLast()	Removes the node at the end of the LinkedList<T> .
ToString()	Returns a string that represents the current object. (Inherited from Object)

Explicit Interface Implementations

[] [Expand table](#)

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array , starting at a particular Array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .

ICollection<T>.Add(T)	Adds an item at the end of the ICollection<T> .
ICollection<T>.IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through the linked list as a collection.
IEnumerable<T>.GetEnumerator()	Returns an enumerator that iterates through a collection.

Extension Methods

[\[+\] Expand table](#)

TolImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
TolImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.

<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .

<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSouce>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(I Enumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(I Enumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(I Enumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(I Enumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(I Enumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(I Enumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(I Enumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(I Enumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(I Enumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(I Enumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(I Enumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform

	function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by

	using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified

	<p>index in a sequence.</p>
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the

	sequence contains no elements.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Groups the elements of a sequence according to a specified key selector function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>,	Groups the elements of a sequence according to a specified

<code>Func<TKey, IEnumerable<TElement>, TResult>()</code>	key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.

<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IQueryable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IQueryable<TSource>, Func<TSource, Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IQueryable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IQueryable<TSource>, Func<TSource, Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IQueryable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IQueryable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource, Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource, Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource, Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IQueryable<TSource>, Func<TSource, Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Decimal>)</code>	Invokes a transform function on each element of a sequence and

	returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Invokes a transform function on each element of a generic

	sequence and returns the minimum resulting value.
MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Sorts the elements of a sequence in descending order according to a key.
Prepend<TSource>(IEnumerable<TSource>, TSource)	Adds a value to the beginning of the sequence.
Reverse<TSource>(IEnumerable<TSource>)	Inverts the order of the elements in a sequence.
Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Projects each element of a sequence into a new form.
Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)	Projects each element of a sequence into a new form by incorporating the element's index.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> ,

	<p>and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.</p>
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.

<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>Skip<TSource>(IEnumerable<TSource>, Int32)</code>	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
<code>SkipLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Computes the sum of the sequence of <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.

<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToListAsync<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer and an element selector function.
<code>ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Lookup<TKey,TElement></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)</code>	Attempts to determine the number of elements in a sequence without forcing an enumeration.
<code>Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set union of two sequences by using a specified <code>IEqualityComparer<T></code> .
<code>Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set union of two sequences by using the default equality comparer.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.

Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the

	descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

This type is not thread safe. If the [LinkedList<T>](#) needs to be accessed by multiple threads, you will need to implement your own synchronization mechanism.

A [LinkedList<T>](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. In the rare case where an enumeration contends with write accesses, the collection must be locked during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

See also

- [LinkedListNode<T>](#)

LinkedList<T> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [LinkedList<T>](#) class.

Overloads

[] [Expand table](#)

LinkedList<T>()	Initializes a new instance of the LinkedList<T> class that is empty.
LinkedList<T>(IEnumerable<T>)	Initializes a new instance of the LinkedList<T> class that contains elements copied from the specified IEnumerable and has sufficient capacity to accommodate the number of elements copied.
LinkedList<T>(SerializationInfo, StreamingContext)	Initializes a new instance of the LinkedList<T> class that is serializable with the specified SerializationInfo and StreamingContext .

LinkedList<T>()

Initializes a new instance of the [LinkedList<T>](#) class that is empty.

C#

```
public LinkedList();
```

Examples

The following code example creates and initializes a [LinkedList<T>](#) of type [String](#), adds several nodes, and then displays its contents.

C#

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

```

public class GenericCollection
{
    public static void Main()
    {
        // Create and initialize a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();
        ll.AddLast("red");
        ll.AddLast("orange");
        ll.AddLast("yellow");
        ll.AddLast("orange");

        // Display the contents of the LinkedList.
        if (ll.Count > 0)
        {
            Console.WriteLine("The first item in the list is {0}.",
ll.First.Value);
            Console.WriteLine("The last item in the list is {0}.",
ll.Last.Value);

            Console.WriteLine("The LinkedList contains:");
            foreach (String s in ll)
                Console.WriteLine("    {0}", s);
        }
        else
        {
            Console.WriteLine("The LinkedList is empty.");
        }
    }
}

/* This code produces the following output.

The first item in the list is red.
The last item in the list is orange.
The LinkedList contains:
    red
    orange
    yellow
    orange
*/

```

Remarks

`LinkedList<T>` accepts `null` as a valid `Value` for reference types and allows duplicate values.

If the `LinkedList<T>` is empty, the `First` and `Last` properties contain `null`.

This constructor is an O(1) operation.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>(IEnumerable<T>)

Initializes a new instance of the [LinkedList<T>](#) class that contains elements copied from the specified [IEnumerable](#) and has sufficient capacity to accommodate the number of elements copied.

C#

```
public LinkedList(System.Collections.Generic.IEnumerable<T> collection);
```

Parameters

collection [IEnumerable<T>](#)

The [IEnumerable](#) whose elements are copied to the new [LinkedList<T>](#).

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Examples

For an example that includes this constructor, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If `collection` has no elements then the new [LinkedList<T>](#) is empty, and the [First](#) and [Last](#) properties contain `null`.

This constructor is an O(n) operation, where n is the number of elements in collection.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>(SerializationInfo, StreamingContext)

Initializes a new instance of the [LinkedList<T>](#) class that is serializable with the specified [SerializationInfo](#) and [StreamingContext](#).

C#

```
protected LinkedList(System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object containing the information required to serialize the [LinkedList<T>](#).

context [StreamingContext](#)

A [StreamingContext](#) object containing the source and destination of the serialized stream associated with the [LinkedList<T>](#).

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the [First](#) and [Last](#) properties contain `null`.

This constructor is an O(n) operation.

See also

- [System.Runtime.Serialization](#)

Applies to

▼ .NET 10 and other versions

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

LinkedList<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of nodes actually contained in the [LinkedList<T>](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of nodes actually contained in the [LinkedList<T>](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Examples

For an example that includes this property, see the [LinkedList<T>](#) class.

Remarks

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.First Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the first node of the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T>? First { get; }
```

Property Value

[LinkedListNode<T>](#)

The first [LinkedListNode<T>](#) of the [LinkedList<T>](#).

Examples

For an example that includes this property, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the [First](#) and [Last](#) properties contain `null`.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.Last Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the last node of the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T>? Last { get; }
```

Property Value

[LinkedListNode<T>](#)

The last [LinkedListNode<T>](#) of the [LinkedList<T>](#).

Examples

For an example that includes this property, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the [First](#) and [Last](#) properties contain `null`.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.AddAfter Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds a new node or value after an existing node in the [LinkedList<T>](#).

Overloads

[] [Expand table](#)

AddAfter(LinkedListNode<T>, LinkedListNode<T>)	Adds the specified new node after the specified existing node in the LinkedList<T> .
AddAfter(LinkedListNode<T>, T)	Adds a new node containing the specified value after the specified existing node in the LinkedList<T> .

AddAfter(LinkedListNode<T>, LinkedListNode<T>)

Adds the specified new node after the specified existing node in the [LinkedList<T>](#).

C#

```
public void AddAfter(System.Collections.Generic.LinkedListNode<T> node,  
System.Collections.Generic.LinkedListNode<T> newNode);
```

Parameters

node [LinkedListNode<T>](#)

The [LinkedListNode<T>](#) after which to insert `newNode`.

newNode [LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) to add to the [LinkedList<T>](#).

Exceptions

[ArgumentNullException](#)

`node` is `null`.

-or-

`newNode` is `null`.

`InvalidOperationException`

`node` is not in the current `LinkedList<T>`.

-or-

`newNode` belongs to another `LinkedList<T>`.

Examples

For an example that includes this method, see the `LinkedList<T>` class.

Remarks

`LinkedList<T>` accepts `null` as a valid `Value` for reference types and allows duplicate values.

This method is an O(1) operation.

See also

- [AddBefore](#)
- [AddFirst](#)
- [AddLast](#)
- [Remove](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

AddAfter(LinkedListNode<T>, T)

Adds a new node containing the specified value after the specified existing node in the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T>
AddAfter(System.Collections.Generic.LinkedListNode<T> node, T value);
```

Parameters

node [LinkedListNode<T>](#)

The [LinkedListNode<T>](#) after which to insert a new [LinkedListNode<T>](#) containing **value**.

value [T](#)

The value to add to the [LinkedList<T>](#).

Returns

[LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) containing **value**.

Exceptions

[ArgumentNullException](#)

node is `null`.

[InvalidOperationException](#)

node is not in the current [LinkedList<T>](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

This method is an O(1) operation.

See also

- [AddBefore](#)
- [AddFirst](#)
- [AddLast](#)
- [Remove](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.AddBefore Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds a new node or value before an existing node in the [LinkedList<T>](#).

Overloads

[+] [Expand table](#)

AddBefore(LinkedListNode<T>, LinkedListNode<T>)	Adds the specified new node before the specified existing node in the LinkedList<T> .
AddBefore(LinkedListNode<T>, T)	Adds a new node containing the specified value before the specified existing node in the LinkedList<T> .

AddBefore(LinkedListNode<T>, LinkedListNode<T>)

Adds the specified new node before the specified existing node in the [LinkedList<T>](#).

C#

```
public void AddBefore(System.Collections.Generic.LinkedListNode<T> node,  
System.Collections.Generic.LinkedListNode<T> newNode);
```

Parameters

node [LinkedListNode<T>](#)

The [LinkedListNode<T>](#) before which to insert `newNode`.

newNode [LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) to add to the [LinkedList<T>](#).

Exceptions

[ArgumentNullException](#)

`node` is `null`.

-or-

`newNode` is `null`.

InvalidOperationException

`node` is not in the current [LinkedList<T>](#).

-or-

`newNode` belongs to another [LinkedList<T>](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

This method is an O(1) operation.

See also

- [AddAfter](#)
- [AddFirst](#)
- [AddLast](#)
- [Remove](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

AddBefore(LinkedListNode<T>, T)

Adds a new node containing the specified value before the specified existing node in the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T>
AddBefore(System.Collections.Generic.LinkedListNode<T> node, T value);
```

Parameters

node [LinkedListNode<T>](#)

The [LinkedListNode<T>](#) before which to insert a new [LinkedListNode<T>](#) containing **value**.

value [T](#)

The value to add to the [LinkedList<T>](#).

Returns

[LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) containing **value**.

Exceptions

[ArgumentNullException](#)

node is `null`.

[InvalidOperationException](#)

node is not in the current [LinkedList<T>](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

This method is an O(1) operation.

See also

- [AddAfter](#)
- [AddFirst](#)
- [AddLast](#)
- [Remove](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.AddFirst Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds a new node or value at the start of the [LinkedList<T>](#).

Overloads

[] [Expand table](#)

AddFirst(LinkedListNode<T>)	Adds the specified new node at the start of the LinkedList<T> .
AddFirst(T)	Adds a new node containing the specified value at the start of the LinkedList<T> .

AddFirst(LinkedListNode<T>)

Adds the specified new node at the start of the [LinkedList<T>](#).

C#

```
public void AddFirst(System.Collections.Generic.LinkedListNode<T> node);
```

Parameters

node [LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) to add at the start of the [LinkedList<T>](#).

Exceptions

[ArgumentNullException](#)

node is `null`.

[InvalidOperationException](#)

node belongs to another [LinkedList<T>](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the new node becomes the [First](#) and the [Last](#).

This method is an O(1) operation.

See also

- [AddAfter](#)
- [AddBefore](#)
- [AddLast](#)
- [RemoveFirst\(\)](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

AddFirst(T)

Adds a new node containing the specified value at the start of the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T> AddFirst(T value);
```

Parameters

value `T`

The value to add at the start of the [LinkedList<T>](#).

Returns

[LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) containing `value`.

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the new node becomes the [First](#) and the [Last](#).

This method is an O(1) operation.

See also

- [AddAfter](#)
- [AddBefore](#)
- [AddLast](#)
- [RemoveFirst\(\)](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.AddLast Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds a new node or value at the end of the [LinkedList<T>](#).

Overloads

[] [Expand table](#)

AddLast(LinkedListNode<T>)	Adds the specified new node at the end of the LinkedList<T> .
AddLast(T)	Adds a new node containing the specified value at the end of the LinkedList<T> .

AddLast(LinkedListNode<T>)

Adds the specified new node at the end of the [LinkedList<T>](#).

C#

```
public void AddLast(System.Collections.Generic.LinkedListNode<T> node);
```

Parameters

node [LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) to add at the end of the [LinkedList<T>](#).

Exceptions

[ArgumentNullException](#)

node is `null`.

[InvalidOperationException](#)

node belongs to another [LinkedList<T>](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the new node becomes the [First](#) and the [Last](#).

This method is an O(1) operation.

See also

- [AddAfter](#)
- [AddBefore](#)
- [AddFirst](#)
- [RemoveLast\(\)](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

AddLast(T)

Adds a new node containing the specified value at the end of the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T> AddLast(T value);
```

Parameters

value `T`

The value to add at the end of the [LinkedList<T>](#).

Returns

[LinkedListNode<T>](#)

The new [LinkedListNode<T>](#) containing `value`.

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the new node becomes the [First](#) and the [Last](#).

This method is an O(1) operation.

See also

- [AddAfter](#)
- [AddBefore](#)
- [AddFirst](#)
- [RemoveLast\(\)](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all nodes from the [LinkedList<T>](#).

C#

```
public void Clear();
```

Implements

[Clear\(\)](#)

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[Count](#) is set to zero, and references to other objects from elements of the collection are also released. [First](#) and [Last](#) are set to `null`.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove](#)
- [RemoveFirst\(\)](#)
- [RemoveLast\(\)](#)

LinkedList<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a value is in the [LinkedList<T>](#).

C#

```
public bool Contains(T value);
```

Parameters

value T

The value to locate in the [LinkedList<T>](#). The value can be `null` for reference types.

Returns

Boolean

`true` if `value` is found in the [LinkedList<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

This method performs a linear search; therefore, this method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Find\(T\)](#)
- [FindLast\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

LinkedList<T>.CopyTo(T[], Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the entire [LinkedList<T>](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

C#

```
public void CopyTo(T[] array, int index);
```

Parameters

array [T\[\]](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [LinkedList<T>](#).

The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

The number of elements in the source [LinkedList<T>](#) is greater than the available space from [index](#) to the end of the destination [array](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [LinkedList<T>](#).

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.Find(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Finds the first node that contains the specified value.

C#

```
public System.Collections.Generic.LinkedListNode<T>? Find(T value);
```

Parameters

value [T](#)

The value to locate in the [LinkedList<T>](#).

Returns

[LinkedListNode<T>](#)

The first [LinkedListNode<T>](#) that contains the specified value, if found; otherwise, [null](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

The [LinkedList<T>](#) is searched forward starting at [First](#) and ending at [Last](#).

This method performs a linear search; therefore, this method is an $O(n)$ operation, where [n](#) is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [FindLast\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

LinkedList<T>.FindLast(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Finds the last node that contains the specified value.

C#

```
public System.Collections.Generic.LinkedListNode<T>? FindLast(T value);
```

Parameters

value **T**

The value to locate in the [LinkedList<T>](#).

Returns

[LinkedListNode<T>](#)

The last [LinkedListNode<T>](#) that contains the specified value, if found; otherwise, `null`.

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

The [LinkedList<T>](#) is searched backward starting at [Last](#) and ending at [First](#).

This method performs a linear search; therefore, this method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Find\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

LinkedList<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedList<T>.Enumerator GetEnumerator();
```

Returns

[LinkedList<T>.Enumerator](#)

An [LinkedList<T>.Enumerator](#) for the [LinkedList<T>](#).

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [LinkedList<T>.Enumerator](#)
- [IEnumerator<T>](#)

LinkedList<T>.GetObjectData(SerializationInfo, StreamingContext) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and returns the data needed to serialize the [LinkedList<T>](#) instance.

C#

```
public virtual void GetObjectData(System.Runtime.Serialization.SerializationInfo
info, System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information required to serialize the [LinkedList<T>](#) instance.

context [StreamingContext](#)

A [StreamingContext](#) object that contains the source and destination of the serialized stream associated with the [LinkedList<T>](#) instance.

Implements

[GetObjectData\(SerializationInfo, StreamingContext\)](#)

Exceptions

[ArgumentNullException](#)

`info` is `null`.

Remarks

This method is an O(`n`) operation, where `n` is [Count](#).

Applies to

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

See also

- [ISerializable](#)
- [SerializationInfo](#)
- [StreamingContext](#)
- [OnDeserialization\(Object\)](#)

LinkedList<T>.OnDeserialization(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and raises the deserialization event when the deserialization is complete.

C#

```
public virtual void OnDeserialization(object? sender);
```

Parameters

sender [Object](#)

The source of the deserialization event.

Implements

[OnDeserialization\(Object\)](#)

Exceptions

[SerializationException](#)

The [SerializationInfo](#) object associated with the current [LinkedList<T>](#) instance is invalid.

Remarks

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	2.0, 2.1

See also

- [ISerializable](#)
- [SerializationInfo](#)
- [StreamingContext](#)
- [GetObjectData\(SerializationInfo, StreamingContext\)](#)

LinkedList<T>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a node or value from the [LinkedList<T>](#).

Overloads

[] [Expand table](#)

Remove(LinkedListNode<T>)	Removes the specified node from the LinkedList<T> .
Remove(T)	Removes the first occurrence of the specified value from the LinkedList<T> .

Remove(LinkedListNode<T>)

Removes the specified node from the [LinkedList<T>](#).

C#

```
public void Remove(System.Collections.Generic.LinkedListNode<T> node);
```

Parameters

node [LinkedListNode<T>](#)

The [LinkedListNode<T>](#) to remove from the [LinkedList<T>](#).

Exceptions

[ArgumentNullException](#)

node is `null`.

[InvalidOperationException](#)

node is not in the current [LinkedList<T>](#).

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

This method is an O(1) operation.

See also

- [RemoveFirst\(\)](#)
- [RemoveLast\(\)](#)
- [Clear\(\)](#)
- [AddBefore](#)
- [AddAfter](#)
- [AddFirst](#)
- [AddLast](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Remove(T)

Removes the first occurrence of the specified value from the [LinkedList<T>](#).

C#

```
public bool Remove(T value);
```

Parameters

value `T`

The value to remove from the [LinkedList<T>](#).

Returns

[Boolean](#)

`true` if the element containing `value` is successfully removed; otherwise, `false`. This method also returns `false` if `value` was not found in the original [LinkedList<T>](#).

Implements

[Remove\(T\)](#)

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

This method performs a linear search; therefore, this method is an $O(n)$ operation, where `n` is [Count](#).

See also

- [RemoveFirst\(\)](#)
- [RemoveLast\(\)](#)
- [Clear\(\)](#)
- [AddBefore](#)
- [AddAfter](#)
- [AddFirst](#)
- [AddLast](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.RemoveFirst Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the node at the start of the [LinkedList<T>](#).

C#

```
public void RemoveFirst();
```

Exceptions

[InvalidOperationException](#)

The [LinkedList<T>](#) is empty.

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove](#)

- RemoveLast()
- Clear()
- AddFirst

LinkedList<T>.RemoveLast Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the node at the end of the [LinkedList<T>](#).

C#

```
public void RemoveLast();
```

Exceptions

[InvalidOperationException](#)

The [LinkedList<T>](#) is empty.

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove](#)

- RemoveFirst()
- Clear()
- AddLast

LinkedList<T>.ICollection<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item at the end of the [ICollection<T>](#).

C#

```
void ICollection<T>.Add(T value);
```

Parameters

value **T**

The value to add at the end of the [ICollection<T>](#).

Implements

[Add\(T\)](#)

Examples

For an example that includes this method, see the [LinkedList<T>](#) class.

Remarks

[LinkedList<T>](#) accepts `null` as a valid [Value](#) for reference types and allows duplicate values.

If the [LinkedList<T>](#) is empty, the new node becomes the [First](#) and the [Last](#).

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [AddLast](#)

LinkedList<T>.ICollection<T>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool System.Collections.Generic.ICollection<T>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [LinkedList<T>](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.IEnumerable<T>.Get Enumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerator<T> IEnumerable<T>.GetEnumerator();
```

Returns

[IEnumerator<T>](#)

An [IEnumerator<T>](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

LinkedList<T>.ICollection.CopyTo(Array, Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the non-generic implementations of [ICollection.CopyTo](#) throw [InvalidOperationException](#), whereas the generic implementations throw [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedList<T>.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [LinkedList<T>](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#).

Synchronization is effective only if all threads lock this object before accessing the collection.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

LinkedList<T>.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [LinkedList<T>](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#).

Synchronization is effective only if all threads lock this object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;  
lock (ic.SyncRoot) {
```

```
// Access the collection.  
}
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

LinkedList<T>.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the linked list as a collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the linked list as a collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetEnumerator\(\)](#)
- [GetInternalEnumerator\(\)](#)
- [IEnumerator](#)

LinkedListNode<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a node in a [LinkedList<T>](#). This class cannot be inherited.

C#

```
public sealed class LinkedListNode<T>
```

Type Parameters

T

Specifies the element type of the linked list.

Inheritance [Object](#) → [LinkedListNode<T>](#)

Examples

The following code example creates a [LinkedListNode<T>](#), adds it to a [LinkedList<T>](#), and tracks the values of its properties as the [LinkedList<T>](#) changes.

C#

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

public class GenericCollection {
    public static void Main() {
        // Create a new LinkedListNode of type String and displays its properties.
        LinkedListNode<String> lln = new LinkedListNode<String>( "orange" );
        Console.WriteLine( "After creating the node ...." );
        DisplayProperties( lln );

        // Create a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();

        // Add the "orange" node and display its properties.
        ll.AddLast( lln );
        Console.WriteLine( "After adding the node to the empty LinkedList ...." );
    }

    void DisplayProperties( LinkedListNode<String> lln ) {
        Console.WriteLine( "Value: " + lln.Value );
        Console.WriteLine( "Next: " + lln.Next );
    }
}
```

```

        DisplayProperties( lln );

        // Add nodes before and after the "orange" node and display the "orange"
        node's properties.
        ll.AddFirst( "red" );
        ll.AddLast( "yellow" );
        Console.WriteLine( "After adding red and yellow ...." );
        DisplayProperties( lln );
    }

    public static void DisplayProperties( LinkedListNode<String> lln ) {
        if ( lln.List == null )
            Console.WriteLine( "    Node is not linked." );
        else
            Console.WriteLine( "    Node belongs to a linked list with {0} elements.",
lln.List.Count );

        if ( lln.Previous == null )
            Console.WriteLine( "    Previous node is null." );
        else
            Console.WriteLine( "    Value of previous node: {0}", lln.Previous.Value
);

        Console.WriteLine( "    Value of current node: {0}", lln.Value );

        if ( lln.Next == null )
            Console.WriteLine( "    Next node is null." );
        else
            Console.WriteLine( "    Value of next node: {0}", lln.Next.Value );

        Console.WriteLine();
    }
}

```

/*

This code produces the following output.

After creating the node
 Node is not linked.
 Previous node is null.
 Value of current node: orange
 Next node is null.

After adding the node to the empty LinkedList
 Node belongs to a linked list with 1 elements.
 Previous node is null.
 Value of current node: orange
 Next node is null.

After adding red and yellow
 Node belongs to a linked list with 3 elements.
 Value of previous node: red
 Value of current node: orange

```
Value of next node: yellow
```

```
*/
```

Remarks

Each element of the [LinkedList<T>](#) collection is a [LinkedListNode<T>](#). The [LinkedListNode<T>](#) contains a value, a reference to the [LinkedList<T>](#) that it belongs to, a reference to the next node, and a reference to the previous node.

Constructors

[] [Expand table](#)

LinkedListNode<T> (T)	Initializes a new instance of the LinkedListNode<T> class, containing the specified value.
---	--

Properties

[] [Expand table](#)

List	Gets the LinkedList<T> that the LinkedListNode<T> belongs to.
Next	Gets the next node in the LinkedList<T> .
Previous	Gets the previous node in the LinkedList<T> .
Value	Gets the value contained in the node.
ValueRef	Gets a reference to the value held by the node.

Methods

[] [Expand table](#)

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
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](#))

[ToString\(\)](#) Returns a string that represents the current object.
(Inherited from [Object](#))

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedListNode<T>(T) Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [LinkedListNode<T>](#) class, containing the specified value.

C#

```
public LinkedListNode(T value);
```

Parameters

value T

The value to contain in the [LinkedListNode<T>](#).

Examples

The following code example creates a [LinkedListNode<T>](#), adds it to a [LinkedList<T>](#), and tracks the values of its properties as the [LinkedList<T>](#) changes.

C#

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

public class GenericCollection {
    public static void Main() {
        // Create a new LinkedListNode of type String and displays its properties.
        LinkedListNode<String> lln = new LinkedListNode<String>( "orange" );
        Console.WriteLine( "After creating the node ...." );
        DisplayProperties( lln );

        // Create a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();

        // Add the "orange" node and display its properties.
        ll.AddLast( lln );
        Console.WriteLine( "After adding the node to the empty LinkedList ...." );
        DisplayProperties( lln );

        // Add nodes before and after the "orange" node and display the "orange"
    }
}
```

```

node's properties.

    ll.AddFirst( "red" );
    ll.AddLast( "yellow" );
    Console.WriteLine( "After adding red and yellow ...." );
    DisplayProperties( lln );
}

public static void DisplayProperties( LinkedListNode<String> lln ) {
    if ( lln.List == null )
        Console.WriteLine( "    Node is not linked." );
    else
        Console.WriteLine( "    Node belongs to a linked list with {0} elements.",
lln.List.Count );

    if ( lln.Previous == null )
        Console.WriteLine( "    Previous node is null." );
    else
        Console.WriteLine( "    Value of previous node: {0}", lln.Previous.Value
);

    Console.WriteLine( "    Value of current node: {0}", lln.Value );

    if ( lln.Next == null )
        Console.WriteLine( "    Next node is null." );
    else
        Console.WriteLine( "    Value of next node: {0}", lln.Next.Value );

    Console.WriteLine();
}
}

/*

```

This code produces the following output.

After creating the node

```

Node is not linked.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding the node to the empty LinkedList

```

Node belongs to a linked list with 1 elements.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding red and yellow

```

Node belongs to a linked list with 3 elements.
Value of previous node: red
Value of current node: orange
Value of next node: yellow

```

*/

Remarks

The [List](#), [Next](#), and [Previous](#) properties are set to `null`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedListNode<T>.List Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the [LinkedList<T>](#) that the [LinkedListNode<T>](#) belongs to.

C#

```
public System.Collections.Generic.LinkedList<T>? List { get; }
```

Property Value

[LinkedList<T>](#)

A reference to the [LinkedList<T>](#) that the [LinkedListNode<T>](#) belongs to, or `null` if the [LinkedListNode<T>](#) is not linked.

Examples

The following code example creates a [LinkedListNode<T>](#), adds it to a [LinkedList<T>](#), and tracks the values of its properties as the [LinkedList<T>](#) changes.

C#

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

public class GenericCollection {
    public static void Main() {
        // Create a new LinkedListNode of type String and displays its properties.
        LinkedListNode<String> lln = new LinkedListNode<String>( "orange" );
        Console.WriteLine( "After creating the node ...." );
        DisplayProperties( lln );

        // Create a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();

        // Add the "orange" node and display its properties.
        ll.AddLast( lln );
        Console.WriteLine( "After adding the node to the empty LinkedList ...." );
        DisplayProperties( lln );
    }

    private void DisplayProperties( LinkedListNode<String> node ) {
        Console.WriteLine( "Value: " + node.Value );
        Console.WriteLine( "Next: " + node.Next );
    }
}
```

```

    // Add nodes before and after the "orange" node and display the "orange"
node's properties.
    ll.AddFirst( "red" );
    ll.AddLast( "yellow" );
    Console.WriteLine( "After adding red and yellow ...." );
    DisplayProperties( lln );
}

public static void DisplayProperties( LinkedListNode<String> lln ) {
    if ( lln.List == null )
        Console.WriteLine( "    Node is not linked." );
    else
        Console.WriteLine( "    Node belongs to a linked list with {0} elements.",
lln.List.Count );

    if ( lln.Previous == null )
        Console.WriteLine( "    Previous node is null." );
    else
        Console.WriteLine( "    Value of previous node: {0}", lln.Previous.Value
);

    Console.WriteLine( "    Value of current node: {0}", lln.Value );

    if ( lln.Next == null )
        Console.WriteLine( "    Next node is null." );
    else
        Console.WriteLine( "    Value of next node: {0}", lln.Next.Value );

    Console.WriteLine();
}
}

/*

```

This code produces the following output.

After creating the node

```

Node is not linked.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding the node to the empty LinkedList

```

Node belongs to a linked list with 1 elements.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding red and yellow

```

Node belongs to a linked list with 3 elements.
Value of previous node: red
Value of current node: orange
Value of next node: yellow

```

*/

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedListNode<T>.Next Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the next node in the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T>? Next { get; }
```

Property Value

[LinkedListNode<T>](#)

A reference to the next node in the [LinkedList<T>](#), or `null` if the current node is the last element ([Last](#)) of the [LinkedList<T>](#).

Examples

The following code example creates a [LinkedListNode<T>](#), adds it to a [LinkedList<T>](#), and tracks the values of its properties as the [LinkedList<T>](#) changes.

C#

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

public class GenericCollection {
    public static void Main() {
        // Create a new LinkedListNode of type String and displays its properties.
        LinkedListNode<String> lln = new LinkedListNode<String>( "orange" );
        Console.WriteLine( "After creating the node ...." );
        DisplayProperties( lln );

        // Create a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();

        // Add the "orange" node and display its properties.
        ll.AddLast( lln );
        Console.WriteLine( "After adding the node to the empty LinkedList ...." );
        DisplayProperties( lln );
    }

    private void DisplayProperties( LinkedListNode<String> node ) {
        Console.WriteLine( "Value: " + node.Value );
        Console.WriteLine( "Next: " + node.Next );
    }
}
```

```

    // Add nodes before and after the "orange" node and display the "orange"
node's properties.
    ll.AddFirst( "red" );
    ll.AddLast( "yellow" );
    Console.WriteLine( "After adding red and yellow ...." );
    DisplayProperties( lln );
}

public static void DisplayProperties( LinkedListNode<String> lln ) {
    if ( lln.List == null )
        Console.WriteLine( "    Node is not linked." );
    else
        Console.WriteLine( "    Node belongs to a linked list with {0} elements.",
lln.List.Count );

    if ( lln.Previous == null )
        Console.WriteLine( "    Previous node is null." );
    else
        Console.WriteLine( "    Value of previous node: {0}", lln.Previous.Value
);

    Console.WriteLine( "    Value of current node: {0}", lln.Value );

    if ( lln.Next == null )
        Console.WriteLine( "    Next node is null." );
    else
        Console.WriteLine( "    Value of next node: {0}", lln.Next.Value );

    Console.WriteLine();
}
}

/*

```

This code produces the following output.

After creating the node

```

Node is not linked.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding the node to the empty LinkedList

```

Node belongs to a linked list with 1 elements.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding red and yellow

```

Node belongs to a linked list with 3 elements.
Value of previous node: red
Value of current node: orange
Value of next node: yellow

```

*/

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedListNode<T>.Previous Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the previous node in the [LinkedList<T>](#).

C#

```
public System.Collections.Generic.LinkedListNode<T>? Previous { get; }
```

Property Value

[LinkedListNode<T>](#)

A reference to the previous node in the [LinkedList<T>](#), or `null` if the current node is the first element ([First](#)) of the [LinkedList<T>](#).

Examples

The following code example creates a [LinkedListNode<T>](#), adds it to a [LinkedList<T>](#), and tracks the values of its properties as the [LinkedList<T>](#) changes.

C#

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

public class GenericCollection {
    public static void Main() {
        // Create a new LinkedListNode of type String and displays its properties.
        LinkedListNode<String> lln = new LinkedListNode<String>( "orange" );
        Console.WriteLine( "After creating the node ...." );
        DisplayProperties( lln );

        // Create a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();

        // Add the "orange" node and display its properties.
        ll.AddLast( lln );
        Console.WriteLine( "After adding the node to the empty LinkedList ...." );
        DisplayProperties( lln );
    }

    private void DisplayProperties( LinkedListNode<String> node ) {
        Console.WriteLine( "Value: " + node.Value );
        Console.WriteLine( "Next: " + node.Next );
        Console.WriteLine( "Previous: " + node.Previous );
    }
}
```

```

    // Add nodes before and after the "orange" node and display the "orange"
node's properties.
    ll.AddFirst( "red" );
    ll.AddLast( "yellow" );
    Console.WriteLine( "After adding red and yellow ...." );
    DisplayProperties( lln );
}

public static void DisplayProperties( LinkedListNode<String> lln ) {
    if ( lln.List == null )
        Console.WriteLine( "    Node is not linked." );
    else
        Console.WriteLine( "    Node belongs to a linked list with {0} elements.",
lln.List.Count );

    if ( lln.Previous == null )
        Console.WriteLine( "    Previous node is null." );
    else
        Console.WriteLine( "    Value of previous node: {0}", lln.Previous.Value
);

    Console.WriteLine( "    Value of current node: {0}", lln.Value );

    if ( lln.Next == null )
        Console.WriteLine( "    Next node is null." );
    else
        Console.WriteLine( "    Value of next node: {0}", lln.Next.Value );

    Console.WriteLine();
}
}

/*

```

This code produces the following output.

After creating the node

```

Node is not linked.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding the node to the empty LinkedList

```

Node belongs to a linked list with 1 elements.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding red and yellow

```

Node belongs to a linked list with 3 elements.
Value of previous node: red
Value of current node: orange
Value of next node: yellow

```

*/

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedListNode<T>.Value Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the value contained in the node.

C#

```
public T Value { get; set; }
```

Property Value

T

The value contained in the node.

Examples

The following code example creates a [LinkedListNode<T>](#), adds it to a [LinkedList<T>](#), and tracks the values of its properties as the [LinkedList<T>](#) changes.

C#

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

public class GenericCollection {
    public static void Main() {
        // Create a new LinkedListNode of type String and displays its properties.
        LinkedListNode<String> lln = new LinkedListNode<String>( "orange" );
        Console.WriteLine( "After creating the node ...." );
        DisplayProperties( lln );

        // Create a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();

        // Add the "orange" node and display its properties.
        ll.AddLast( lln );
        Console.WriteLine( "After adding the node to the empty LinkedList ...." );
        DisplayProperties( lln );
    }

    private void DisplayProperties( LinkedListNode<String> node ) {
        Console.WriteLine( "Value: " + node.Value );
        Console.WriteLine( "Next: " + node.Next );
    }
}
```

```

    // Add nodes before and after the "orange" node and display the "orange"
node's properties.
    ll.AddFirst( "red" );
    ll.AddLast( "yellow" );
    Console.WriteLine( "After adding red and yellow ...." );
    DisplayProperties( lln );
}

public static void DisplayProperties( LinkedListNode<String> lln ) {
    if ( lln.List == null )
        Console.WriteLine( "    Node is not linked." );
    else
        Console.WriteLine( "    Node belongs to a linked list with {0} elements.", 
lln.List.Count );

    if ( lln.Previous == null )
        Console.WriteLine( "    Previous node is null." );
    else
        Console.WriteLine( "    Value of previous node: {0}", lln.Previous.Value
);

    Console.WriteLine( "    Value of current node: {0}", lln.Value );

    if ( lln.Next == null )
        Console.WriteLine( "    Next node is null." );
    else
        Console.WriteLine( "    Value of next node:      {0}", lln.Next.Value );

    Console.WriteLine();
}
}

/*

```

This code produces the following output.

After creating the node

```

Node is not linked.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding the node to the empty LinkedList

```

Node belongs to a linked list with 1 elements.
Previous node is null.
Value of current node: orange
Next node is null.

```

After adding red and yellow

```

Node belongs to a linked list with 3 elements.
Value of previous node: red
Value of current node: orange
Value of next node:      yellow

```

*/

Remarks

This property is set in the [LinkedListNode<T>](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

LinkedListNode<T>.ValueRef Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a reference to the value held by the node.

C#

```
public ref T ValueRef { get; }
```

Property Value

T

A reference to the value held by the node.

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

List<T>.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [List<T>](#).

C#

```
public struct List<T>.Enumerator : System.Collections.Generic.IEnumerator<T>
```

Type Parameters

T

Inheritance [Object](#) → [ValueType](#) → List<T>.Enumerator

Implements [IEnumerator<T>](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned

`false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to `MoveNext` or `IEnumerator.Reset` throws an `InvalidOperationException`.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in `System.Collections.Generic` are not synchronized.

Properties

[+] Expand table

<code>Current</code>	Gets the element at the current position of the enumerator.
----------------------	---

Methods

[+] Expand table

<code>Dispose()</code>	Releases all resources used by the <code>List<T>.Enumerator</code> .
<code>MoveNext()</code>	Advances the enumerator to the next element of the <code>List<T></code> .

Explicit Interface Implementations

[+] Expand table

<code>IEnumerator.Current</code>	Gets the element at the current position of the enumerator.
<code>IEnumerator.Reset()</code>	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

List<T>.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the [List<T>](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

List<T>.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [List<T>.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [List<T>](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to [MoveNext](#) advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is

irrecoverably invalidated and the next call to [MoveNext](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

List<T>.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the [List<T>](#) at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

`IEnumerator.Current` does not move the position of the enumerator, and consecutive calls to `IEnumerator.Current` return the same object until either `MoveNext` or `IEnumerator.Reset` is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

List<T>.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling [IEnumator.Reset](#), you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [IEnumator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [Current](#)
- [IEnumerator](#)

List<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

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

C#

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

Type Parameters

T

The type of elements in the list.

Inheritance [Object](#) → List<T>

Derived [System.Data.Services.ExpandSegmentCollection](#)

[System.Workflow.Activities.OperationParameterInfoCollection](#)

[System.Workflow.Activities.WorkflowRoleCollection](#)

[System.Workflow.ComponentModel.ActivityCollection](#)

[System.Workflow.ComponentModel.Design.ActivityDesignerGlyphCollection](#)

More...

Implements [ICollection<T>](#) , [IEnumerable<T>](#) , [IList<T>](#) , [IReadOnlyCollection<T>](#) ,

[IReadOnlyList<T>](#) , [ICollection](#) , [IEnumerable](#) , [IList](#)

Remarks

For more information about this API, see [Supplemental API remarks for List<T>](#).

Constructors

Expand table

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.

Properties

 Expand table

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

 Expand table

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, IComparer<T>)	Searches the entire 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.
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[], Int32)	Copies the entire 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.
EnsureCapacity(Int32)	Ensures that the capacity of this list is at least the specified <code>capacity</code> . If the current capacity is less than <code>capacity</code> , it is increased to at least the specified <code>capacity</code> .
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, 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.
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)	Searches for the specified object and returns the zero-based index of the first occurrence within the entire List<T> .
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, 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.
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)	Searches for the specified object and returns the zero-based index of the last occurrence within the entire List<T> .
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.
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.

Explicit Interface Implementations

[] [Expand table](#)

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array , starting at a particular Array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
ICollection<T>.IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through a collection.

<code>IEnumerable<T>.GetEnumerator()</code>	Returns an enumerator that iterates through a collection.
<code>IList.Add(Object)</code>	Adds an item to the <code>IList</code> .
<code>IList.Contains(Object)</code>	Determines whether the <code>IList</code> contains a specific value.
<code>IList.IndexOf(Object)</code>	Determines the index of a specific item in the <code>IList</code> .
<code>IList.Insert(Int32, Object)</code>	Inserts an item to the <code>IList</code> at the specified index.
<code>IList.IsFixedSize</code>	Gets a value indicating whether the <code>IList</code> has a fixed size.
<code>IList.IsReadOnly</code>	Gets a value indicating whether the <code>IList</code> is read-only.
<code>IList.Item[Int32]</code>	Gets or sets the element at the specified index.
<code>IList.Remove(Object)</code>	Removes the first occurrence of a specific object from the <code>IList</code> .

Extension Methods

[] [Expand table](#)

<code>ToImmutableArray<TSource>(IEnumerable<TSource>)</code>	Creates an immutable array from the specified collection.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.

<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input

	<code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.
<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the average of a sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.

Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most <code>size</code> .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a

	specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set difference of two sequences according to a specified key selector function.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set difference of two sequences according to a specified key selector function.
First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element in a sequence that satisfies a specified condition.
First<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence.
FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.

<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.

<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.

<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic

	sequence and returns the maximum resulting value.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function.
Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Invokes a transform function on each element of a sequence and

	returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in descending order according to a key.
Prepend<TSource>(IEnumerable<TSource>, TSource)	Adds a value to the beginning of the sequence.

<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified <code>IEqualityComparer<T></code> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.

Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.

SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a IEnumerable<T> .
<code>ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key

	selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to specified key selector and element selector functions.

TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .

Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.

[Remove<T>\(IEnumerable<T>\)](#)

Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`Shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

It is safe to perform multiple read operations on a [List<T>](#), but issues can occur if the collection is modified while it's being read. To ensure thread safety, lock the collection during a read or write operation. To enable a collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization. For collections with built-in synchronization, see the classes in the [System.Collections.Concurrent](#) namespace. For an inherently thread-safe alternative, see the [ImmutableList<T>](#) class.

See also

- [IList](#)
- [ImmutableList<T>](#)
- [Performing Culture-Insensitive String Operations in Collections](#)
- [Iterators \(C#\)](#)
- [Iterators \(Visual Basic\)](#)

List<T> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [List<T>](#) class.

Overloads

[+] [Expand table](#)

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.

List<T>()

Initializes a new instance of the [List<T>](#) class that is empty and has the default initial capacity.

C#

```
public List();
```

Examples

The following example demonstrates the parameterless constructor of the [List<T>](#) generic class. The parameterless constructor creates a list with the default capacity, as demonstrated by displaying the [Capacity](#) property.

The example adds, inserts, and removes items, showing how the capacity changes as these methods are used.

C#

```
List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
dinosaurs.Insert(2, "Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

// Shows accessing the list using the Item property.
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosaurs.Remove("Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);
```

```
/* This code example produces the following output:

Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8
Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosaurs[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

TrimExcess()
Capacity: 5
Count: 5

Clear()
Capacity: 5
Count: 0
*/
```

Remarks

The capacity of a [List<T>](#) is the number of elements that the [List<T>](#) can hold. As elements are added to a [List<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, using the [List<T>\(Int32\)](#) constructor and specifying the initial capacity eliminates the need to perform a number of resizing

operations while adding elements to the [List<T>](#).

The capacity can be decreased by calling the [TrimExcess](#) method or by setting the [Capacity](#) property explicitly. Decreasing the capacity reallocates memory and copies all the elements in the [List<T>](#).

This constructor is an O(1) operation.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public List(System.Collections.Generic.IEnumerable<T> collection);
```

Parameters

collection [IEnumerable<T>](#)

The collection whose elements are copied to the new list.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Examples

The following example demonstrates the `List<T>` constructor and various methods of the `List<T>` class that act on ranges. An array of strings is created and passed to the constructor, populating the list with the elements of the array. The `Capacity` property is then displayed, to show that the initial capacity is exactly what is required to hold the input elements.

C#

```
Console.WriteLine("\nInsertRange(3, input)");
dinosaurs.InsertRange(3, input);

Console.WriteLine();
foreach( string dinosaur in dinosaurs )
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\noutput = dinosaurs.GetRange(2, 3).ToArray()");
string[] output = dinosaurs.GetRange(2, 3).ToArray();

Console.WriteLine();
foreach( string dinosaur in output )
{
    Console.WriteLine(dinosaur);
}
}
```

/ This code example produces the following output:*

Capacity: 3

Brachiosaurus
Amargasaurus
Mamenchisaurus

AddRange(dinosaurs)

Brachiosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Amargasaurus
Mamenchisaurus

RemoveRange(2, 2)

Brachiosaurus
Amargasaurus
Amargasaurus
Mamenchisaurus

InsertRange(3, input)

Brachiosaurus
Amargasaurus
Amargasaurus
Tyrannosaurus
Deinonychus
Velociraptor
Mamenchisaurus

```
output = dinosaurs.GetRange(2, 3).ToArray()

Amargasaurus
Tyrannosaurus
Deinonychus
*/
```

Remarks

The elements are copied onto the [List<T>](#) in the same order they are read by the enumerator of the collection.

This constructor is an $O(n)$ operation, where n is the number of elements in `collection`.

See also

- [IEnumerable<T>](#)
- [Capacity](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>(Int32)

Initializes a new instance of the [List<T>](#) class that is empty and has the specified initial capacity.

C#

```
public List(int capacity);
```

Parameters

capacity Int32

The number of elements that the new list can initially store.

Exceptions

ArgumentOutOfRangeException

`capacity` is less than 0.

Examples

The following example demonstrates the `List<T>(Int32)` constructor. A `List<T>` of strings with a capacity of 4 is created, because the ultimate size of the list is known to be exactly 4. The list is populated with four strings, and a read-only copy is created by using the `AsReadOnly` method.

C#

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

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>(4);

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

        dinosaurs.Add("Tyrannosaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Deinonychus");

        Console.WriteLine();
        foreach(string s in dinosaurs)
        {
            Console.WriteLine(s);
        }

        Console.WriteLine("\nIList<string> roDinosaurs =
dinosaurs.AsReadOnly()");
        IList<string> roDinosaurs = dinosaurs.AsReadOnly();

        Console.WriteLine("\nElements in the read-only IList:");
        foreach(string dinosaur in roDinosaurs)
        {
            Console.WriteLine(dinosaur);
        }
    }
}
```

```

Console.WriteLine("\ndinosaurs[2] = \"Coelophysis\"");
dinosaurs[2] = "Coelophysis";

Console.WriteLine("\nElements in the read-only IList:");
foreach(string dinosaur in roDinosaurs)
{
    Console.WriteLine(dinosaur);
}
}

/* This code example produces the following output:

Capacity: 4

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus

IList<string> roDinosaurs = dinosaurs.AsReadOnly()

Elements in the read-only IList:
Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus

dinosaurs[2] = "Coelophysis"

Elements in the read-only IList:
Tyrannosaurus
Amargasaurus
Coelophysis
Deinonychus
*/

```

Remarks

The capacity of a [List<T>](#) is the number of elements that the [List<T>](#) can hold. As elements are added to a [List<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [List<T>](#).

The capacity can be decreased by calling the [TrimExcess](#) method or by setting the [Capacity](#) property explicitly. Decreasing the capacity reallocates memory and copies all the elements in the [List<T>](#).

This constructor is an O(1) operation.

See also

- [Capacity](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.Capacity Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the total number of elements the internal data structure can hold without resizing.

C#

```
public int Capacity { get; set; }
```

Property Value

[Int32](#)

The number of elements that the [List<T>](#) can contain before resizing is required.

Exceptions

[ArgumentOutOfRangeException](#)

`Capacity` is set to a value that is less than [Count](#).

[OutOfMemoryException](#)

There is not enough memory available on the system.

Examples

The following example demonstrates how to check the capacity and count of a [List<T>](#) that contains a simple business object, and illustrates using the [TrimExcess](#) method to remove extra capacity.

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify a part
// but the part name be different for the same Id.
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }
    public int PartId { get; set; }
```

```

public override string ToString()
{
    return "ID: " + PartId + "    Name: " + PartName;
}
public override bool Equals(object obj)
{
    if (obj == null) return false;
    Part objAsPart = obj as Part;
    if (objAsPart == null) return false;
    else return Equals(objAsPart);
}
public override int GetHashCode()
{
    return base.GetHashCode();
}
public bool Equals(Part other)
{
    if (other == null) return false;
    return (this.PartId.Equals(other.PartId));
}
// Should also override == and != operators.
}
public class Example
{

    public static void Main()
    {
        List<Part> parts = new List<Part>();

        Console.WriteLine("\nCapacity: {0}", parts.Capacity);

        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        Console.WriteLine("\nCapacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);

        parts.TrimExcess();
        Console.WriteLine("\nTrimExcess()");
        Console.WriteLine("Capacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);

        parts.Clear();
        Console.WriteLine("\nClear()");
        Console.WriteLine("Capacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);
    }
}

```

```

}

/*
    This code example produces the following output.
        Capacity: 0

        ID: 1234    Name: crank arm
        ID: 1334    Name: chain ring
        ID: 1434    Name: seat
        ID: 1534    Name: cassette
        ID: 1634    Name: shift lever

        Capacity: 8
        Count: 5

        TrimExcess()
        Capacity: 5
        Count: 5

        Clear()
        Capacity: 5
        Count: 0
*/
}

```

The following example shows the [Capacity](#) property at several points in the life of a list. The parameterless constructor is used to create a list of strings with a capacity of 0, and the [Capacity](#) property is displayed to demonstrate this. After the [Add](#) method has been used to add several items, the items are listed, and then the [Capacity](#) property is displayed again, along with the [Count](#) property, to show that the capacity has been increased as needed.

The [Capacity](#) property is displayed again after the [TrimExcess](#) method is used to reduce the capacity to match the count. Finally, the [Clear](#) method is used to remove all items from the list, and the [Capacity](#) and [Count](#) properties are displayed again.

C#

```

List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

```

```
Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
dinosaurs.Insert(2, "Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

// Shows accessing the list using the Item property.
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosaurs.Remove("Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

/* This code example produces the following output:
```

```
Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8
Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
```

```
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosaurs[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

TrimExcess()
Capacity: 5
Count: 5

Clear()
Capacity: 5
Count: 0
*/
```

Remarks

Capacity is the number of elements that the [List<T>](#) can store before resizing is required, whereas **Count** is the number of elements that are actually in the [List<T>](#).

Capacity is always greater than or equal to **Count**. If **Count** exceeds **Capacity** while adding elements, the capacity is increased by automatically reallocating the internal array before copying the old elements and adding the new elements.

If the capacity is significantly larger than the count and you want to reduce the memory used by the [List<T>](#), you can decrease capacity by calling the [TrimExcess](#) method or by setting the **Capacity** property explicitly to a lower value. When the value of **Capacity** is set explicitly, the internal array is also reallocated to accommodate the specified capacity, and all the elements are copied.

Retrieving the value of this property is an O(1) operation; setting the property is an O(*n*) operation, where *n* is the new capacity.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Count](#)

List<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements contained in the [List<T>](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [List<T>](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Examples

The following example demonstrates how to check the capacity and count of a [List<T>](#) that contains a simple business object, and illustrates using the [TrimExcess](#) method to remove extra capacity.

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify a part
// but the part name be different for the same Id.
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }
    public int PartId { get; set; }
    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
```

```

        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{

    public static void Main()
    {
        List<Part> parts = new List<Part>();

        Console.WriteLine("\nCapacity: {0}", parts.Capacity);

        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        Console.WriteLine("\nCapacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);

        parts.TrimExcess();
        Console.WriteLine("\nTrimExcess()");
        Console.WriteLine("Capacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);

        parts.Clear();
        Console.WriteLine("\nClear()");
        Console.WriteLine("Capacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);
    }
    /*

```

This code example produces the following output.

Capacity: 0

ID: 1234 Name: crank arm

```

        ID: 1334  Name: chain ring
        ID: 1434  Name: seat
        ID: 1534  Name: cassette
        ID: 1634  Name: shift lever

    Capacity: 8
    Count: 5

    TrimExcess()
    Capacity: 5
    Count: 5

    Clear()
    Capacity: 5
    Count: 0
}

*/
}

```

The following example shows the value of the [Count](#) property at various points in the life of a list. After the list has been created and populated and its elements displayed, the [Capacity](#) and [Count](#) properties are displayed. These properties are displayed again after the [TrimExcess](#) method has been called, and again after the contents of the list are cleared.

C#

```

List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
dinosaurs.Insert(2, "Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{

```

```
Console.WriteLine(dinosaur);
}

// Shows accessing the list using the Item property.
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosaurs.Remove("Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

/* This code example produces the following output:
```

Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8
Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosaurs[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus

```
Mamenchisaurus  
Deinonychus  
Compsognathus
```

```
TrimExcess()  
Capacity: 5  
Count: 5
```

```
Clear()  
Capacity: 5  
Count: 0  
*/
```

Remarks

Capacity is the number of elements that the [List<T>](#) can store before resizing is required.

Count is the number of elements that are actually in the [List<T>](#).

Capacity is always greater than or equal to **Count**. If **Count** exceeds **Capacity** while adding elements, the capacity is increased by automatically reallocating the internal array before copying the old elements and adding the new elements.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Capacity](#)

List<T>.Item[Int32] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the element at the specified index.

C#

```
public T this[int index] { get; set; }
```

Parameters

index [Int32](#)

The zero-based index of the element to get or set.

Property Value

T

The element at the specified index.

Implements

[Item\[Int32\]](#)

Exceptions

[ArgumentOutOfRangeException](#)

index is less than 0.

-or-

index is equal to or greater than [Count](#).

Examples

The example in this section demonstrates the [Item\[\]](#) property (the indexer in C#) and various other properties and methods of the [List<T>](#) generic class. After the list has been created and populated using the [Add](#) method, an element is retrieved and displayed using the [Item\[\]](#)

property. (For an example that uses the [Item\[\]](#) property to set the value of a list element, see [AsReadOnly](#).)

! Note

Visual Basic, C#, and C++ all have syntax for accessing the [Item\[\]](#) property without using its name. Instead, the variable containing the [List<T>](#) is used as if it were an array.

The C# language uses the [this](#) keyword to define the indexers instead of implementing the [Item\[\]](#) property. Visual Basic implements [Item\[\]](#) as a default property, which provides the same indexing functionality.

C#

```
List<string> dinosaurs = new List<string>();  
  
Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);  
  
dinosaurs.Add("Tyrannosaurus");  
dinosaurs.Add("Amargasaurus");  
dinosaurs.Add("Mamenchisaurus");  
dinosaurs.Add("Deinonychus");  
dinosaurs.Add("Compsognathus");
```

C#

```
// Shows accessing the list using the Item property.  
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);
```

Remarks

[List<T>](#) accepts `null` as a valid value for reference types and allows duplicate elements.

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[index]`.

Retrieving the value of this property is an O(1) operation; setting the property is also an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Count](#)

List<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an object to the end of the [List<T>](#).

C#

```
public void Add(T item);
```

Parameters

item T

The object to be added to the end of the [List<T>](#). The value can be `null` for reference types.

Implements

[Add\(T\)](#)

Examples

The following example demonstrates how to add, remove, and insert a simple business object in a [List<T>](#).

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify the type of part
// but the part name can change.
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
```

```

    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public override int GetHashCode()
    {
        return PartId;
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        // Write out the parts in the list. This will call the overridden ToString
method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Check the list for part #1734. This calls the IEquatable.Equals method
        // of the Part class, which checks the PartId for equality.
        Console.WriteLine("\nContains(\"1734\"): {0}",
        parts.Contains(new Part { PartId = 1734, PartName = "" }));

        // Insert a new item at position 2.
        Console.WriteLine("\nInsert(2, \"1834\")");
        parts.Insert(2, new Part() { PartName = "brake lever", PartId = 1834 });

        //Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }
    }
}

```

```
Console.WriteLine("\nParts[3]: {0}", parts[3]);  
  
Console.WriteLine("\nRemove(\"1534\");  
  
// This will remove part 1534 even though the PartName is different,  
// because the Equals method only checks PartId for equality.  
parts.Remove(new Part() { PartId = 1534, PartName = "cogs" });  
  
Console.WriteLine();  
foreach (Part aPart in parts)  
{  
    Console.WriteLine(aPart);  
}  
Console.WriteLine("\nRemoveAt(3)");  
// This will remove the part at index 3.  
parts.RemoveAt(3);  
  
Console.WriteLine();  
foreach (Part aPart in parts)  
{  
    Console.WriteLine(aPart);  
}  
  
/*  
  
ID: 1234  Name: crank arm  
ID: 1334  Name: chain ring  
ID: 1434  Name: regular seat  
ID: 1444  Name: banana seat  
ID: 1534  Name: cassette  
ID: 1634  Name: shift lever  
  
Contains("1734"): False  
  
Insert(2, "1834")  
ID: 1234  Name: crank arm  
ID: 1334  Name: chain ring  
ID: 1834  Name: brake lever  
ID: 1434  Name: regular seat  
ID: 1444  Name: banana seat  
ID: 1534  Name: cassette  
ID: 1634  Name: shift lever  
  
Parts[3]: ID: 1434  Name: regular seat  
  
Remove("1534")  
  
ID: 1234  Name: crank arm  
ID: 1334  Name: chain ring  
ID: 1834  Name: brake lever  
ID: 1434  Name: regular seat  
ID: 1444  Name: banana seat  
ID: 1634  Name: shift lever
```

```

        RemoveAt(3)

        ID: 1234  Name: crank arm
        ID: 1334  Name: chain ring
        ID: 1834  Name: brake lever
        ID: 1444  Name: banana seat
        ID: 1634  Name: shift lever

    */
}

}

```

The following example demonstrates several properties and methods of the `List<T>` generic class, including the `Add` method. The parameterless constructor is used to create a list of strings with a capacity of 0. The `Capacity` property is displayed, and then the `Add` method is used to add several items. The items are listed, and the `Capacity` property is displayed again, along with the `Count` property, to show that the capacity has been increased as needed.

Other properties and methods are used to search for, insert, and remove elements from the list, and finally to clear the list.

C#

```

List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
dinosaurs.Insert(2, "Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{

```

```
Console.WriteLine(dinosaur);
}

// Shows accessing the list using the Item property.
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosaurs.Remove("Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

/* This code example produces the following output:
```

Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8
Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosaurs[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus

```
Mamenchisaurus  
Deinonychus  
Compsognathus
```

```
TrimExcess()  
Capacity: 5  
Count: 5  
  
Clear()  
Capacity: 5  
Count: 0  
*/
```

Remarks

`List<T>` accepts `null` as a valid value for reference types and allows duplicate elements.

If `Count` already equals `Capacity`, the capacity of the `List<T>` is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If `Count` is less than `Capacity`, this method is an $O(1)$ operation. If the capacity needs to be increased to accommodate the new element, this method becomes an $O(n)$ operation, where n is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [AddRange\(IEnumerable<T>\)](#)
- [Insert\(Int32, T\)](#)
- [Remove\(T\)](#)
- [Count](#)

List<T>.AddRange(IEnumerable<T>)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds the elements of the specified collection to the end of the [List<T>](#).

C#

```
public void AddRange(System.Collections.Generic.IEnumerable<T> collection);
```

Parameters

collection [IEnumerable<T>](#)

The collection whose elements should be added to the end of the [List<T>](#). The collection itself cannot be `null`, but it can contain elements that are `null`, if type `T` is a reference type.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Examples

The following example demonstrates the [AddRange](#) method and various other methods of the [List<T>](#) class that act on ranges. An array of strings is created and passed to the constructor, populating the list with the elements of the array. The [AddRange](#) method is called, with the list as its argument. The result is that the current elements of the list are added to the end of the list, duplicating all the elements.

C#

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

public class Example
{
    public static void Main()
    {
```

```
string[] input = { "Brachiosaurus",
                  "Amargasaurus",
                  "Mamenchisaurus" };

List<string> dinosaurs = new List<string>(input);

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

Console.WriteLine();
foreach( string dinosaur in dinosaurs )
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nAddRange(dinosaurs)");
dinosaurs.AddRange(dinosaurs);

Console.WriteLine();
foreach( string dinosaur in dinosaurs )
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nRemoveRange(2, 2)");
dinosaurs.RemoveRange(2, 2);

Console.WriteLine();
foreach( string dinosaur in dinosaurs )
{
    Console.WriteLine(dinosaur);
}

input = new string[] { "Tyrannosaurus",
                      "Deinonychus",
                      "Velociraptor" };

Console.WriteLine("\nInsertRange(3, input)");
dinosaurs.InsertRange(3, input);

Console.WriteLine();
foreach( string dinosaur in dinosaurs )
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\noutput = dinosaurs.GetRange(2, 3).ToArray()");
string[] output = dinosaurs.GetRange(2, 3).ToArray();

Console.WriteLine();
foreach( string dinosaur in output )
{
    Console.WriteLine(dinosaur);
}
}
```

```
/* This code example produces the following output:

Capacity: 3

Brachiosaurus
Amargasaurus
Mamenchisaurus

AddRange(dinosaurs)

Brachiosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Amargasaurus
Mamenchisaurus

RemoveRange(2, 2)

Brachiosaurus
Amargasaurus
Amargasaurus
Mamenchisaurus

InsertRange(3, input)

Brachiosaurus
Amargasaurus
Amargasaurus
Tyrannosaurus
Deinonychus
Velociraptor
Mamenchisaurus

output = dinosaurs.GetRange(2, 3).ToArray()

Amargasaurus
Tyrannosaurus
Deinonychus
*/
```

Remarks

The order of the elements in the collection is preserved in the [List<T>](#).

If the new [Count](#) (the current [Count](#) plus the size of the collection) will be greater than [Capacity](#), the capacity of the [List<T>](#) is increased by automatically reallocating the internal array to accommodate the new elements, and the existing elements are copied to the new array before the new elements are added.

If the [List<T>](#) can accommodate the new elements without increasing the [Capacity](#), this method is an $O(n)$ operation, where n is the number of elements to be added. If the capacity needs to be increased to accommodate the new elements, this method becomes an $O(n + m)$ operation, where n is the number of elements to be added and m is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerable<T>](#)
- [Capacity](#)
- [Count](#)
- [Add\(T\)](#)
- [InsertRange\(Int32, IEnumerable<T>\)](#)
- [RemoveRange\(Int32, Int32\)](#)
- [GetRange\(Int32, Int32\)](#)

List<T>.AsReadOnly Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns a read-only [ReadOnlyCollection<T>](#) wrapper for the current collection.

C#

```
public System.Collections.ObjectModel.ReadOnlyCollection<T> AsReadOnly();
```

Returns

[ReadOnlyCollection<T>](#)

An object that acts as a read-only wrapper around the current [List<T>](#).

Examples

The following example demonstrates the [AsReadOnly](#) method. A [List<T>](#) of strings with a capacity of 4 is created, because the ultimate size of the list is known to be exactly 4. The list is populated with four strings, and the [AsReadOnly](#) method is used to get a read-only [IList<T>](#) generic interface implementation that wraps the original list.

An element of the original list is set to "Coelophysis" using the [Item\[\]](#) property (the indexer in C#), and the contents of the read-only list are displayed again to demonstrate that it is just a wrapper for the original list.

C#

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

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>(4);

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

        dinosaurs.Add("Tyrannosaurus");
        dinosaurs.Add("Amargasaurus");
```

```
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");

Console.WriteLine();
foreach(string s in dinosaurs)
{
    Console.WriteLine(s);
}

Console.WriteLine("\nIList<string> roDinosaurs = dinosaurs.AsReadOnly()");
IList<string> roDinosaurs = dinosaurs.AsReadOnly();

Console.WriteLine("\nElements in the read-only IList:");
foreach(string dinosaur in roDinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\ndinosaurs[2] = \"Coelophysis\"");
dinosaurs[2] = "Coelophysis";

Console.WriteLine("\nElements in the read-only IList:");
foreach(string dinosaur in roDinosaurs)
{
    Console.WriteLine(dinosaur);
}
}
```

/* This code example produces the following output:

Capacity: 4

```
Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
```

```
IList<string> roDinosaurs = dinosaurs.AsReadOnly()
```

Elements in the read-only IList:

```
Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
```

```
dinosaurs[2] = "Coelophysis"
```

Elements in the read-only IList:

```
Tyrannosaurus
Amargasaurus
Coelophysis
Deinonychus
*/
```

Remarks

To prevent any modifications to the [List<T>](#) object, expose it only through this wrapper. A [ReadOnlyCollection<T>](#) object does not expose methods that modify the collection. However, if changes are made to the underlying [List<T>](#) object, the read-only collection reflects those changes.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.BinarySearch Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Uses a binary search algorithm to locate a specific element in the sorted [List<T>](#) or a portion of it.

Overloads

 [Expand table](#)

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.
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.

C#

```
public int BinarySearch(T item);
```

Parameters

item [T](#)

The object to locate. The value can be `null` for reference types.

Returns

[Int32](#)

The zero-based index of `item` in the sorted `List<T>`, if `item` is found; otherwise, a negative number that is the bitwise complement of the index of the next element that is larger than `item` or, if there is no larger element, the bitwise complement of `Count`.

Exceptions

InvalidOperationException

The default comparer `Default` cannot find an implementation of the `IComparable<T>` generic interface or the `IComparable` interface for type `T`.

Examples

The following example demonstrates the `Sort()` method overload and the `BinarySearch(T)` method overload. A `List<T>` of strings is created and populated with four strings, in no particular order. The list is displayed, sorted, and displayed again.

The `BinarySearch(T)` method overload is then used to search for two strings that are not in the list, and the `Insert` method is used to insert them. The return value of the `BinarySearch(T)` method is negative in each case, because the strings are not in the list. Taking the bitwise complement (the `~` operator in C#, `Xor -1` in Visual Basic) of this negative number produces the index of the first element in the list that is larger than the search string, and inserting at this location preserves the sort order. The second search string is larger than any element in the list, so the insertion position is at the end of the list.

C#

```
List<string> dinosaurs = new List<string>();

dinosaurs.Add("Pachycephalosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");

Console.WriteLine("Initial list:");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nSort:");
dinosaurs.Sort();

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
```

```
}
```

```
Console.WriteLine("\nBinarySearch and Insert \"Coelophysis\"");
```

```
int index = dinosaurs.BinarySearch("Coelophysis");
```

```
if (index < 0)
```

```
{
```

```
    dinosaurs.Insert(~index, "Coelophysis");
```

```
}
```

```
Console.WriteLine();
```

```
foreach(string dinosaur in dinosaurs)
```

```
{
```

```
    Console.WriteLine(dinosaur);
```

```
}
```

```
Console.WriteLine("\nBinarySearch and Insert \"Tyrannosaurus\"");
```

```
index = dinosaurs.BinarySearch("Tyrannosaurus");
```

```
if (index < 0)
```

```
{
```

```
    dinosaurs.Insert(~index, "Tyrannosaurus");
```

```
}
```

```
Console.WriteLine();
```

```
foreach(string dinosaur in dinosaurs)
```

```
{
```

```
    Console.WriteLine(dinosaur);
```

```
}
```

```
/* This code example produces the following output:
```

Initial list:

Pachycephalosaurus
Amargasaurus
Mamenchisaurus
Deinonychus

Sort:

Amargasaurus
Deinonychus
Mamenchisaurus
Pachycephalosaurus

BinarySearch and Insert "Coelophysis":

Amargasaurus
Coelophysis
Deinonychus
Mamenchisaurus
Pachycephalosaurus

BinarySearch and Insert "Tyrannosaurus":

Amargasaurus
Coelophysis

```
Deinonychus
Mamenchisaurus
Pachycephalosaurus
Tyrannosaurus
*/
```

Remarks

This method uses the default comparer [Comparer<T>.Default](#) for type `T` to determine the order of list elements. The [Comparer<T>.Default](#) property checks whether type `T` implements the [IComparable<T>](#) generic interface and uses that implementation, if available. If not, [Comparer<T>.Default](#) checks whether type `T` implements the [IComparable](#) interface. If type `T` does not implement either interface, [Comparer<T>.Default](#) throws an [InvalidOperationException](#).

The [List<T>](#) must already be sorted according to the comparer implementation; otherwise, the result is incorrect.

Comparing `null` with any reference type is allowed and does not generate an exception when using the [IComparable<T>](#) generic interface. When sorting, `null` is considered to be less than any other object.

If the [List<T>](#) contains more than one element with the same value, the method returns only one of the occurrences, and it might return any one of the occurrences, not necessarily the first one.

If the [List<T>](#) does not contain the specified value, the method returns a negative integer. You can apply the bitwise complement operation (`~`) to this negative integer to get the index of the first element that is larger than the search value. When inserting the value into the [List<T>](#), this index should be used as the insertion point to maintain the sort order.

This method is an $O(\log n)$ operation, where n is the number of elements in the range.

See also

- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int BinarySearch(T item, System.Collections.Generic.IComparer<T>?
comparer);
```

Parameters

item [T](#)

The object to locate. The value can be `null` for reference types.

comparer [IComparer<T>](#)

The [IComparer<T>](#) implementation to use when comparing elements.

-or-

`null` to use the default comparer [Default](#).

Returns

[Int32](#)

The zero-based index of `item` in the sorted [List<T>](#), if `item` is found; otherwise, a negative number that is the bitwise complement of the index of the next element that is larger than `item` or, if there is no larger element, the bitwise complement of [Count](#).

Exceptions

[InvalidOperationException](#)

`comparer` is `null`, and the default comparer `Default` cannot find an implementation of the `IComparable<T>` generic interface or the `IComparable` interface for type `T`.

Examples

The following example demonstrates the `Sort(IComparer<T>)` method overload and the `BinarySearch(T, IComparer<T>)` method overload.

The example defines an alternative comparer for strings named `DinoCompare`, which implements the `IComparer<string>` (`IComparer(Of String)` in Visual Basic) generic interface. The comparer works as follows: First, the comparands are tested for `null`, and a null reference is treated as less than a non-null. Second, the string lengths are compared, and the longer string is deemed to be greater. Third, if the lengths are equal, ordinary string comparison is used.

A `List<T>` of strings is created and populated with four strings, in no particular order. The list is displayed, sorted using the alternate comparer, and displayed again.

The `BinarySearch(T, IComparer<T>)` method overload is then used to search for several strings that are not in the list, employing the alternate comparer. The `Insert` method is used to insert the strings. These two methods are located in the function named `SearchAndInsert`, along with code to take the bitwise complement (the `~` operator in C#, `Xor -1` in Visual Basic) of the negative number returned by `BinarySearch(T, IComparer<T>)` and use it as an index for inserting the new string.

C#

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

public class DinoComparer: IComparer<string>
{
    public int Compare(string x, string y)
    {
        if (x == null)
        {
            if (y == null)
            {
                // If x is null and y is null, they're
                // equal.
                return 0;
            }
            else
            {
                // If x is null and y is not null, y
                // is greater.
                return -1;
            }
        }
        else
        {
            if (y == null)
            {
                // If x is not null and y is null, x
                // is greater.
                return 1;
            }
            else
            {
                // If both are not null, compare their
                // lengths.
                if (x.Length < y.Length)
                    return -1;
                else if (x.Length > y.Length)
                    return 1;
                else
                    return 0;
            }
        }
    }
}
```

```

        }
    }
    else
    {
        // If x is not null...
        //
        if (y == null)
            // ...and y is null, x is greater.
        {
            return 1;
        }
        else
        {
            // ...and y is not null, compare the
            // lengths of the two strings.
            //
            int retval = x.Length.CompareTo(y.Length);

            if (retval != 0)
            {
                // If the strings are not of equal length,
                // the longer string is greater.
                //
                return retval;
            }
            else
            {
                // If the strings are of equal length,
                // sort them with ordinary string comparison.
                //
                return x.CompareTo(y);
            }
        }
    }
}

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();
        dinosaurs.Add("Pachycephalosaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Deinonychus");
        Display(dinosaurs);

        DinoComparer dc = new DinoComparer();

        Console.WriteLine("\nSort with alternate comparer:");
        dinosaurs.Sort(dc);
        Display(dinosaurs);

        SearchAndInsert(dinosaurs, "Coelophysis", dc);
    }
}

```

```

        Display(dinosaurs);

        SearchAndInsert(dinosaurs, "Oviraptor", dc);
        Display(dinosaurs);

        SearchAndInsert(dinosaurs, "Tyrannosaur", dc);
        Display(dinosaurs);

        SearchAndInsert(dinosaurs, null, dc);
        Display(dinosaurs);
    }

    private static void SearchAndInsert(List<string> list,
        string insert, DinoComparer dc)
    {
        Console.WriteLine("\nBinarySearch and Insert \"{0}\":", insert);

        int index = list.BinarySearch(insert, dc);

        if (index < 0)
        {
            list.Insert(~index, insert);
        }
    }

    private static void Display(List<string> list)
    {
        Console.WriteLine();
        foreach( string s in list )
        {
            Console.WriteLine(s);
        }
    }
}

```

/ This code example produces the following output:*

Pachycephalosaurus
Amargasaurus
Mamenchisaurus
Deinonychus

Sort with alternate comparer:

Deinonychus
Amargasaurus
Mamenchisaurus
Pachycephalosaurus

BinarySearch and Insert "Coelophysis":

Coelophysis
Deinonychus
Amargasaurus
Mamenchisaurus

```
Pachycephalosaurus
```

```
BinarySearch and Insert "Oviraptor":
```

```
Oviraptor
Coelophysis
Deinonychus
Amargasaurus
Mamenchisaurus
Pachycephalosaurus
```

```
BinarySearch and Insert "Tyrannosaur":
```

```
Oviraptor
Coelophysis
Deinonychus
Tyrannosaur
Amargasaurus
Mamenchisaurus
Pachycephalosaurus
```

```
BinarySearch and Insert "":
```

```
Oviraptor
Coelophysis
Deinonychus
Tyrannosaur
Amargasaurus
Mamenchisaurus
Pachycephalosaurus
*/
```

Remarks

The comparer customizes how the elements are compared. For example, you can use a [CaseInsensitiveComparer](#) instance as the comparer to perform case-insensitive string searches.

If `comparer` is provided, the elements of the `List<T>` are compared to the specified value using the specified [IComparer<T>](#) implementation.

If `comparer` is `null`, the default comparer [Comparer<T>.Default](#) checks whether type `T` implements the [IComparable<T>](#) generic interface and uses that implementation, if available. If not, [Comparer<T>.Default](#) checks whether type `T` implements the [IComparable](#) interface. If type `T` does not implement either interface, [Comparer<T>.Default](#) throws [InvalidOperationException](#).

The `List<T>` must already be sorted according to the comparer implementation; otherwise, the result is incorrect.

Comparing `null` with any reference type is allowed and does not generate an exception when using the `IComparable<T>` generic interface. When sorting, `null` is considered to be less than any other object.

If the `List<T>` contains more than one element with the same value, the method returns only one of the occurrences, and it might return any one of the occurrences, not necessarily the first one.

If the `List<T>` does not contain the specified value, the method returns a negative integer. You can apply the bitwise complement operation (`~`) to this negative integer to get the index of the first element that is larger than the search value. When inserting the value into the `List<T>`, this index should be used as the insertion point to maintain the sort order.

This method is an $O(\log n)$ operation, where n is the number of elements in the range.

See also

- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int BinarySearch(int index, int count, T item,  
System.Collections.Generic.IComparer<T>? comparer);
```

Parameters

index `Int32`

The zero-based starting index of the range to search.

count `Int32`

The length of the range to search.

item `T`

The object to locate. The value can be `null` for reference types.

comparer `IComparer<T>`

The `IComparer<T>` implementation to use when comparing elements, or `null` to use the default comparer `Default`.

Returns

`Int32`

The zero-based index of `item` in the sorted `List<T>`, if `item` is found; otherwise, a negative number that is the bitwise complement of the index of the next element that is larger than `item` or, if there is no larger element, the bitwise complement of `Count`.

Exceptions

[ArgumentOutOfRangeException](#)

`index` is less than 0.

-or-

`count` is less than 0.

[ArgumentException](#)

`index` and `count` do not denote a valid range in the `List<T>`.

[InvalidOperationException](#)

`comparer` is `null`, and the default comparer `Default` cannot find an implementation of the `IComparable<T>` generic interface or the `IComparable` interface for type `T`.

Examples

The following example demonstrates the `Sort(Int32, Int32, IComparer<T>)` method overload and the `BinarySearch(Int32, Int32, T, IComparer<T>)` method overload.

The example defines an alternative comparer for strings named `DinoCompare`, which implements the `IComparer<string>` (`IComparer(Of String)` in Visual Basic) generic interface. The comparer works as follows: First, the comparands are tested for `null`, and a null reference is treated as less than a non-null. Second, the string lengths are compared, and the longer string is deemed to be greater. Third, if the lengths are equal, ordinary string comparison is used.

A `List<T>` of strings is created and populated with the names of five herbivorous dinosaurs and three carnivorous dinosaurs. Within each of the two groups, the names are not in any particular sort order. The list is displayed, the range of herbivores is sorted using the alternate comparer, and the list is displayed again.

The `BinarySearch(Int32, Int32, T, IComparer<T>)` method overload is then used to search only the range of herbivores for "Brachiosaurus". The string is not found, and the bitwise complement (the `~` operator in C#, `Xor -1` in Visual Basic) of the negative number returned by the `BinarySearch(Int32, Int32, T, IComparer<T>)` method is used as an index for inserting the new string.

C#

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

public class DinoComparer: IComparer<string>
{
    public int Compare(string x, string y)
    {
        if (x == null)
        {
            if (y == null)
            {
                // If x is null and y is null, they're
                // equal.
                return 0;
            }
            else
            {
                // If x is null and y is not null, y
                // is greater.
            }
        }
        else
        {
            if (y == null)
            {
                // If x is not null and y is null, x
                // is greater.
            }
            else
            {
                if (x.Length < y.Length)
                    return -1;
                else if (x.Length > y.Length)
                    return 1;
                else
                    return string.Compare(x, y);
            }
        }
    }
}
```

```

        return -1;
    }
}
else
{
    // If x is not null...
    //
    if (y == null)
        // ...and y is null, x is greater.
    {
        return 1;
    }
    else
    {
        // ...and y is not null, compare the
        // lengths of the two strings.
        //
        int retval = x.Length.CompareTo(y.Length);

        if (retval != 0)
        {
            // If the strings are not of equal length,
            // the longer string is greater.
            //
            return retval;
        }
        else
        {
            // If the strings are of equal length,
            // sort them with ordinary string comparison.
            //
            return x.CompareTo(y);
        }
    }
}
}

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        dinosaurs.Add("Pachycephalosaurus");
        dinosaurs.Add("Parasauralophus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Galimimus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Oviraptor");
        dinosaurs.Add("Tyrannosaurus");

        int herbivores = 5;
        Display(dinosaurs);
    }
}
```

```
DinoComparer dc = new DinoComparer();

Console.WriteLine("\nSort a range with the alternate comparer:");
dinosaurs.Sort(0, herbivores, dc);
Display(dinosaurs);

Console.WriteLine("\nBinarySearch a range and Insert \'{0}\':",
    "Brachiosaurus");

int index = dinosaurs.BinarySearch(0, herbivores, "Brachiosaurus", dc);

if (index < 0)
{
    dinosaurs.Insert(~index, "Brachiosaurus");
    herbivores++;
}

Display(dinosaurs);
}

private static void Display(List<string> list)
{
    Console.WriteLine();
    foreach( string s in list )
    {
        Console.WriteLine(s);
    }
}
```

/ This code example produces the following output:*

```
Pachycephalosaurus
Parasauralophus
Amargasaurus
Galimimus
Mamenchisaurus
Deinonychus
Oviraptor
Tyrannosaurus
```

Sort a range with the alternate comparer:

```
Galimimus
Amargasaurus
Mamenchisaurus
Parasauralophus
Pachycephalosaurus
Deinonychus
Oviraptor
Tyrannosaurus
```

BinarySearch a range and Insert "Brachiosaurus":

```
Galimimus
Amargasaurus
Brachiosaurus
Mamenchisaurus
Parasauralophus
Pachycephalosaurus
Deinonychus
Oviraptor
Tyrannosaurus
*/
```

Remarks

The comparer customizes how the elements are compared. For example, you can use a [CaseInsensitiveComparer](#) instance as the comparer to perform case-insensitive string searches.

If `comparer` is provided, the elements of the [List<T>](#) are compared to the specified value using the specified [IComparer<T>](#) implementation.

If `comparer` is `null`, the default comparer [Comparer<T>.Default](#) checks whether type `T` implements the [IComparable<T>](#) generic interface and uses that implementation, if available. If not, [Comparer<T>.Default](#) checks whether type `T` implements the [IComparable](#) interface. If type `T` does not implement either interface, [Comparer<T>.Default](#) throws [InvalidOperationException](#).

The [List<T>](#) must already be sorted according to the comparer implementation; otherwise, the result is incorrect.

Comparing `null` with any reference type is allowed and does not generate an exception when using the [IComparable<T>](#) generic interface. When sorting, `null` is considered to be less than any other object.

If the [List<T>](#) contains more than one element with the same value, the method returns only one of the occurrences, and it might return any one of the occurrences, not necessarily the first one.

If the [List<T>](#) does not contain the specified value, the method returns a negative integer. You can apply the bitwise complement operation (`~`) to this negative integer to get the index of the first element that is larger than the search value. When inserting the value into the [List<T>](#), this index should be used as the insertion point to maintain the sort order.

This method is an $O(\log n)$ operation, where n is the number of elements in the range.

See also

- [IComparer<T>](#)
- [IComparable<T>](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements from the [List<T>](#).

C#

```
public void Clear();
```

Implements

[Clear\(\)](#) , [Clear\(\)](#)

Examples

The following example demonstrates the [Clear](#) method and various other properties and methods of the [List<T>](#) generic class. The [Clear](#) method is used at the end of the program, to remove all items from the list, and the [Capacity](#) and [Count](#) properties are then displayed.

C#

```
List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));
```

```
Console.WriteLine("\nInsert(2, \"Compsognathus\"));  
dinosaurs.Insert(2, "Compsognathus");  
  
Console.WriteLine();  
foreach(string dinosaur in dinosaurs)  
{  
    Console.WriteLine(dinosaur);  
}  
  
// Shows accessing the list using the Item property.  
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);  
  
Console.WriteLine("\nRemove(\"Compsognathus\"));  
dinosaurs.Remove("Compsognathus");  
  
Console.WriteLine();  
foreach(string dinosaur in dinosaurs)  
{  
    Console.WriteLine(dinosaur);  
}  
  
dinosaurs.TrimExcess();  
Console.WriteLine("\nTrimExcess()");  
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);  
Console.WriteLine("Count: {0}", dinosaurs.Count);  
  
dinosaurs.Clear();  
Console.WriteLine("\nClear()");  
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);  
Console.WriteLine("Count: {0}", dinosaurs.Count);
```

/ This code example produces the following output:*

Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8

Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

```
dinosaurs[3]: Mamenchisaurus
```

```
Remove("Compsognathus")
```

```
Tyrannosaurus  
Amargasaurus  
Mamenchisaurus  
Deinonychus  
Compsognathus
```

```
TrimExcess()
```

```
Capacity: 5
```

```
Count: 5
```

```
Clear()
```

```
Capacity: 5
```

```
Count: 0
```

```
*/
```

Remarks

[Count](#) is set to 0, and references to other objects from elements of the collection are also released.

[Capacity](#) remains unchanged. To reset the capacity of the [List<T>](#), call the [TrimExcess](#) method or set the [Capacity](#) property directly. Decreasing the capacity reallocates memory and copies all the elements in the [List<T>](#). Trimming an empty [List<T>](#) sets the capacity of the [List<T>](#) to the default capacity.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [TrimExcess\(\)](#)

- Capacity
- Count

List<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether an element is in the [List<T>](#).

C#

```
public bool Contains(T item);
```

Parameters

item T

The object to locate in the [List<T>](#). The value can be `null` for reference types.

Returns

Boolean

`true` if `item` is found in the [List<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Examples

The following example demonstrates the [Contains](#) and [Exists](#) methods on a [List<T>](#) that contains a simple business object that implements [Equals](#).

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify a part
// but the part name can change.
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }
    public int PartId { get; set; }
```

```

public override string ToString()
{
    return "ID: " + PartId + "    Name: " + PartName;
}
public override bool Equals(object obj)
{
    if (obj == null) return false;
    Part objAsPart = obj as Part;
    if (objAsPart == null) return false;
    else return Equals(objAsPart);
}
public override int GetHashCode()
{
    return PartId;
}
public bool Equals(Part other)
{
    if (other == null) return false;
    return (this.PartId.Equals(other.PartId));
}
// Should also override == and != operators.
}

public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        // Write out the parts in the list. This will call the overridden ToString
method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Check the list for part #1734. This calls the IEquatable.Equals method
        // of the Part class, which checks the PartId for equality.
        Console.WriteLine("\nContains: Part with Id=1734: {0}",
            parts.Contains(new Part { PartId = 1734, PartName = "" }));

        // Find items where name contains "seat".
        Console.WriteLine("\nFind: Part where name contains \"seat\": {0}",
            parts.Find(x => x.PartName.Contains("seat")));
    }
}

```

```

// Check if an item with Id 1444 exists.
Console.WriteLine("\nExists: Part with Id=1444: {0}",
    parts.Exists(x => x.PartId == 1444));

/*This code example produces the following output:

ID: 1234  Name: crank arm
ID: 1334  Name: chain ring
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

Contains: Part with Id=1734: False

Find: Part where name contains "seat": ID: 1434  Name: regular seat

Exists: Part with Id=1444: True
*/
}
}

```

The following example contains a list of complex objects of type `Cube`. The `Cube` class implements the `IEquatable<T>.Equals` method so that two cubes are considered equal if their dimensions are the same. In this example, the `Contains` method returns `true`, because a cube that has the specified dimensions is already in the collection.

C#

```

using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        List<Cube> cubes = new List<Cube>();

        cubes.Add(new Cube(8, 8, 4));
        cubes.Add(new Cube(8, 4, 8));
        cubes.Add(new Cube(8, 6, 4));

        if (cubes.Contains(new Cube(8, 6, 4)))
            Console.WriteLine("An equal cube is already in the collection.");
        else {
            Console.WriteLine("Cube can be added.");
        }

        //Outputs "An equal cube is already in the collection."
    }
}

```

```

}

public class Cube : IEquatable<Cube>
{
    public Cube(int h, int l, int w)
    {
        this.Height = h;
        this.Length = l;
        this.Width = w;
    }

    public int Height { get; set; }
    public int Length { get; set; }
    public int Width { get; set; }

    public bool Equals(Cube other)
    {
        if (this.Height == other.Height && this.Length == other.Length
            && this.Width == other.Width) {
            return true;
        }
        else {
            return false;
        }
    }
}

```

Remarks

This method determines equality by using the default equality comparer, as defined by the object's implementation of the `IEquatable<T>.Equals` method for `T` (the type of values in the list).

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IndexOf\(T\)](#)
- [LastIndexOf\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

List<T>.ConvertAll<TOoutput>(Converter<T,TOoutput>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Converts the elements in the current [List<T>](#) to another type, and returns a list containing the converted elements.

C#

```
public System.Collections.Generic.List<TOoutput> ConvertAll<TOoutput>(Converter<T,TOoutput> converter);
```

Type Parameters

TOoutput

The type of the elements of the target array.

Parameters

converter Converter<T,TOoutput>

A [Converter<TInput,TOoutput>](#) delegate that converts each element from one type to another type.

Returns

List<TOoutput>

A [List<T>](#) of the target type containing the converted elements from the current [List<T>](#).

Exceptions

ArgumentNullException

converter is null.

Examples

The following example defines a method named `PointFToPoint` that converts a `PointF` structure to a `Point` structure. The example then creates a `List<T>` of `PointF` structures, creates a `Converter<PointF, Point>` delegate (`Converter(Of PointF, Point)` in Visual Basic) to represent the `PointFToPoint` method, and passes the delegate to the `ConvertAll` method. The `ConvertAll` method passes each element of the input list to the `PointFToPoint` method and puts the converted elements into a new list of `Point` structures. Both lists are displayed.

C#

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

public class Example
{
    public static void Main()
    {
        List lpf = new List();

        lpf.Add(new PointF(27.8F, 32.62F));
        lpf.Add(new PointF(99.3F, 147.273F));
        lpf.Add(new PointF(7.5F, 1412.2F));

        Console.WriteLine();
        foreach( PointF p in lpf )
        {
            Console.WriteLine(p);
        }

        List lp = lpf.ConvertAll(
            new Converter<PointF, Point>(PointFToPoint));

        Console.WriteLine();
        foreach( Point p in lp )
        {
            Console.WriteLine(p);
        }
    }

    public static Point PointFToPoint(PointF pf)
    {
        return new Point((int) pf.X, (int) pf.Y);
    }
}

/* This code example produces the following output:

{X=27.8, Y=32.62}
{X=99.3, Y=147.273}
{X=7.5, Y=1412.2}

{X=27, Y=32}
```

```
{X=99,Y=147}  
{X=7,Y=1412}  
*/
```

Remarks

The [Converter<TInput,TOutput>](#) is a delegate to a method that converts an object to the target type. The elements of the current [List<T>](#) are individually passed to the [Converter<TInput,TOutput>](#) delegate, and the converted elements are saved in the new [List<T>](#).

The current [List<T>](#) remains unchanged.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

See also

- [Converter<TInput,TOutput>](#)

List<T>.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [List<T>](#) or a portion of it to an array.

Overloads

 [Expand table](#)

CopyTo(T[], Int32)	Copies the entire List<T> to a compatible one-dimensional array, starting at the specified index of the target array.
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.

Examples

The following example demonstrates all three overloads of the [CopyTo](#) method. A [List<T>](#) of strings is created and populated with 5 strings. An empty string array of 15 elements is created, and the [CopyTo\(T\[\]\)](#) method overload is used to copy all the elements of the list to the array beginning at the first element of the array. The [CopyTo\(T\[\], Int32\)](#) method overload is used to copy all the elements of the list to the array beginning at array index 6 (leaving index 5 empty). Finally, the [CopyTo\(Int32, T\[\], Int32, Int32\)](#) method overload is used to copy 3 elements from the list, beginning with index 2, to the array beginning at array index 12 (leaving index 11 empty). The contents of the array are then displayed.

C#

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

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();
```

```
dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Brachiosaurus");
dinosaurs.Add("Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

// Declare an array with 15 elements.
string[] array = new string[15];

dinosaurs.CopyTo(array);
dinosaurs.CopyTo(array, 6);
dinosaurs.CopyTo(2, array, 12, 3);

Console.WriteLine("\nContents of the array:");
foreach(string dinosaur in array)
{
    Console.WriteLine(dinosaur);
}
}

/*
 * This code example produces the following output:
 */

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Compsognathus

Contents of the array:
Tyrannosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Compsognathus

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Compsognathus

Mamenchisaurus
Brachiosaurus
Compsognathus
*/
```

CopyTo(T[], Int32)

Copies the entire [List<T>](#) to a compatible one-dimensional array, starting at the specified index of the target array.

C#

```
public void CopyTo(T[] array, int arrayIndex);
```

Parameters

array `T[]`

The one-dimensional [Array](#) that is the destination of the elements copied from [List<T>](#). The [Array](#) must have zero-based indexing.

arrayIndex `Int32`

The zero-based index in `array` at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#)

`arrayIndex` is less than 0.

[ArgumentException](#)

The number of elements in the source [List<T>](#) is greater than the available space from `arrayIndex` to the end of the destination `array`.

Remarks

This method uses [Array.Copy](#) to copy the elements.

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [List<T>](#).

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public void CopyTo(int index, T[] array, int arrayIndex, int count);
```

Parameters

index [Int32](#)

The zero-based index in the source [List<T>](#) at which copying begins.

array [T\[\]](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [List<T>](#). The [Array](#) must have zero-based indexing.

arrayIndex [Int32](#)

The zero-based index in [array](#) at which copying begins.

count [Int32](#)

The number of elements to copy.

Exceptions

ArgumentNullException

`array` is `null`.

ArgumentOutOfRangeException

`index` is less than 0.

-or-

`arrayIndex` is less than 0.

-or-

`count` is less than 0.

ArgumentException

`index` is equal to or greater than the `Count` of the source `List<T>`.

-or-

The number of elements from `index` to the end of the source `List<T>` is greater than the available space from `arrayIndex` to the end of the destination `array`.

Remarks

This method uses `Array.Copy` to copy the elements.

The elements are copied to the `Array` in the same order in which the enumerator iterates through the `List<T>`.

This method is an $O(n)$ operation, where n is `count`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

CopyTo(T[])

Copies the entire [List<T>](#) to a compatible one-dimensional array, starting at the beginning of the target array.

C#

```
public void CopyTo(T[] array);
```

Parameters

array `T[]`

The one-dimensional [Array](#) that is the destination of the elements copied from [List<T>](#). The [Array](#) must have zero-based indexing.

Exceptions

[ArgumentNullException](#)

`array` is `null`.

[ArgumentException](#)

The number of elements in the source [List<T>](#) is greater than the number of elements that the destination `array` can contain.

Remarks

This method uses [Array.Copy](#) to copy the elements.

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [List<T>](#).

This method is an O(*n*) operation, where *n* is [Count](#).

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.EnsureCapacity(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

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`.

C#

```
public int EnsureCapacity(int capacity);
```

Parameters

capacity [Int32](#)

The minimum capacity to ensure.

Returns

[Int32](#)

The new capacity of this list.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

List<T>.Exists(Predicate<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [List<T>](#) contains elements that match the conditions defined by the specified predicate.

C#

```
public bool Exists(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the elements to search for.

Returns

[Boolean](#)

`true` if the [List<T>](#) contains one or more elements that match the conditions defined by the specified predicate; otherwise, `false`.

Exceptions

[ArgumentNullException](#)

`match` is `null`.

Examples

The following example demonstrates the [Contains](#) and [Exists](#) methods on a [List<T>](#) that contains a simple business object that implements [Equals](#).

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify a part
// but the part name can change.
```

```
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }
    public int PartId { get; set; }

    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public override int GetHashCode()
    {
        return PartId;
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        // Write out the parts in the list. This will call the overridden ToString
        // method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Check the list for part #1734. This calls the IEquatable.Equals method
        // of the Part class, which checks the PartId for equality.
        Console.WriteLine("\nContains: Part with Id=1734: {0}",
            parts.Contains(new Part { PartId = 1734, PartName = "" }));
    }
}
```

```

// Find items where name contains "seat".
Console.WriteLine("\nFind: Part where name contains \"seat\": {0}",
    parts.Find(x => x.PartName.Contains("seat")));

// Check if an item with Id 1444 exists.
Console.WriteLine("\nExists: Part with Id=1444: {0}",
    parts.Exists(x => x.PartId == 1444));

/*This code example produces the following output:

ID: 1234  Name: crank arm
ID: 1334  Name: chain ring
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

Contains: Part with Id=1734: False

Find: Part where name contains "seat": ID: 1434  Name: regular seat

Exists: Part with Id=1444: True
*/
}
}

```

The following example demonstrates the [Exists](#) method and several other methods that use the [Predicate<T>](#) generic delegate.

A [List<T>](#) of strings is created, containing 8 dinosaur names, two of which (at positions 1 and 5) end with "saurus". The example also defines a search predicate method named [EndsWithSaurus](#), which accepts a string parameter and returns a Boolean value indicating whether the input string ends in "saurus".

The [Find](#), [FindLast](#), and [FindAll](#) methods are used to search the list with the search predicate method, and then the [RemoveAll](#) method is used to remove all entries ending with "saurus".

Finally, the [Exists](#) method is called. It traverses the list from the beginning, passing each element in turn to the [EndsWithSaurus](#) method. The search stops and the method returns `true` if the [EndsWithSaurus](#) method returns `true` for any element. The [Exists](#) method returns `false` because all such elements have been removed.

Note

In C# and Visual Basic, it is not necessary to create the `Predicate<string>` delegate (`Predicate(Of String)` in Visual Basic) explicitly. These languages infer the correct delegate from context and create it automatically.

C#

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

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        dinosaurs.Add("Compsognathus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Oviraptor");
        dinosaurs.Add("Velociraptor");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Dilophosaurus");
        dinosaurs.Add("Gallimimus");
        dinosaurs.Add("Triceratops");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nTrueForAll(EndsWithSaurus): {0}",
            dinosaurs.TrueForAll(EndsWithSaurus));

        Console.WriteLine("\nFind(EndsWithSaurus): {0}",
            dinosaurs.Find(EndsWithSaurus));

        Console.WriteLine("\nFindLast(EndsWithSaurus): {0}",
            dinosaurs.FindLast(EndsWithSaurus));

        Console.WriteLine("\nFindAll(EndsWithSaurus):");
        List<string> sublist = dinosaurs.FindAll(EndsWithSaurus);

        foreach(string dinosaur in sublist)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine(
            "\n{0} elements removed by RemoveAll(EndsWithSaurus).",
            dinosaurs.RemoveAll(EndsWithSaurus));

        Console.WriteLine("\nList now contains:");
    }
}
```

```

        foreach(string dinosaur in dinosaurs)
    {
        Console.WriteLine(dinosaur);
    }

    Console.WriteLine("\nExists(EndsWithSaurus): {0}",
        dinosaurs.Exists(EndsWithSaurus));
}

// Search predicate returns true if a string ends in "saurus".
private static bool EndsWithSaurus(String s)
{
    return s.ToLower().EndsWith("saurus");
}
}

/* This code example produces the following output:

Compsognathus
Amargasaurus
Oviraptor
Velociraptor
Deinonychus
Dilophosaurus
Gallimimus
Triceratops

TrueForAll(EndsWithSaurus): False

Find(EndsWithSaurus): Amargasaurus

FindLast(EndsWithSaurus): Dilophosaurus

FindAll(EndsWithSaurus):
Amargasaurus
Dilophosaurus

2 elements removed by RemoveAll(EndsWithSaurus).

List now contains:
Compsognathus
Oviraptor
Velociraptor
Deinonychus
Gallimimus
Triceratops

Exists(EndsWithSaurus): False
*/

```

Remarks

The [Predicate<T>](#) is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current [List<T>](#) are individually passed to the [Predicate<T>](#) delegate, and processing is stopped when a match is found.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Find\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindIndex](#)
- [FindLastIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [TrueForAll\(Predicate<T>\)](#)
- [Predicate<T>](#)

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

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire [List<T>](#).

C#

```
public T? Find(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the element to search for.

Returns

T

The first element that matches the conditions defined by the specified predicate, if found; otherwise, the default value for type [T](#).

Exceptions

[ArgumentNullException](#)

`match` is `null`.

Examples

The following example demonstrates the [Find](#) method on a [List<T>](#) that contains a simple complex object.

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify a part
// but the part name can change.
```

```
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }
    public int PartId { get; set; }

    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public override int GetHashCode()
    {
        return PartId;
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        // Write out the parts in the list. This will call the overridden ToString
        // method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Check the list for part #1734. This calls the IEquatable.Equals method
        // of the Part class, which checks the PartId for equality.
        Console.WriteLine("\nContains: Part with Id=1734: {0}",
            parts.Contains(new Part { PartId = 1734, PartName = "" }));
    }
}
```

```

// Find items where name contains "seat".
Console.WriteLine("\nFind: Part where name contains \"seat\": {0}",
    parts.Find(x => x.PartName.Contains("seat")));

// Check if an item with Id 1444 exists.
Console.WriteLine("\nExists: Part with Id=1444: {0}",
    parts.Exists(x => x.PartId == 1444));

/*This code example produces the following output:

ID: 1234  Name: crank arm
ID: 1334  Name: chain ring
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

Contains: Part with Id=1734: False

Find: Part where name contains "seat": ID: 1434  Name: regular seat

Exists: Part with Id=1444: True
*/
}
}

```

The following example demonstrates the find methods for the `List<T>` class. The example for the `List<T>` class contains `book` objects, of class `Book`, using the data from the [Sample XML File: Books \(LINQ to XML\)](#). The `FillList` method in the example uses [LINQ to XML](#) to parse the values from the XML to property values of the `book` objects.

The following table describes the examples provided for the find methods.

[] [Expand table](#)

Method	Example
Find(Predicate<T>)	Finds a book by an ID using the <code>IDToFind</code> predicate delegate. C# example uses an anonymous delegate.
FindAll(Predicate<T>)	Find all books that whose <code>Genre</code> property is "Computer" using the <code>FindComputer</code> predicate delegate.
FindLast(Predicate<T>)	Finds the last book in the collection that has a publish date before 2001, using the <code>PubBefore2001</code> predicate delegate. C# example uses an anonymous delegate.

Method	Example
FindIndex(Predicate<T>)	Finds the index of first computer book using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Predicate<T>)	Finds the index of the last computer book using the <code>FindComputer</code> predicate delegate.
FindIndex(Int32, Int32, Predicate<T>)	Finds the index of first computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Int32, Int32, Predicate<T>)	Finds the index of last computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace Find
{
    class Program
    {
        private static string IDtoFind = "bk109";

        private static List<Book> Books = new List<Book>();
        public static void Main(string[] args)
        {
            FillList();

            // Find a book by its ID.
            Book result = Books.Find(
                delegate(Book bk)
                {
                    return bk.ID == IDtoFind;
                }
            );
            if (result != null)
            {
                DisplayResult(result, "Find by ID: " + IDtoFind);
            }
            else
            {
                Console.WriteLine("\nNot found: {0}", IDtoFind);
            }

            // Find last book in collection published before 2001.
            result = Books.FindLast(
                delegate(Book bk)
                {
                    DateTime year2001 = new DateTime(2001, 01, 01);
                }
            );
        }
    }
}

```

```
        return bk.Publish_date < year2001;
    });
    if (result != null)
    {
        DisplayResult(result, "Last book in collection published before
2001:");
    }
    else
    {
        Console.WriteLine("\nNot found: {0}", IDtoFind);
    }

    // Find all computer books.
    List<Book> results = Books.FindAll(FindComputer);
    if (results.Count != 0)
    {
        DisplayResults(results, "All computer:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find all books under $10.00.
    results = Books.FindAll(
        delegate(Book bk)
    {
        return bk.Price < 10.00;
    });
    if (results.Count != 0)
    {
        DisplayResults(results, "Books under $10:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find index values.
    Console.WriteLine();
    int ndx = Books.FindIndex(FindComputer);
    Console.WriteLine("Index of first computer book: {0}", ndx);
    ndx = Books.FindLastIndex(FindComputer);
    Console.WriteLine("Index of last computer book: {0}", ndx);

    int mid = Books.Count / 2;
    ndx = Books.FindIndex(mid, mid, FindComputer);
    Console.WriteLine("Index of first computer book in the second half of
the collection: {0}", ndx);

    ndx = Books.FindLastIndex(Books.Count - 1, mid, FindComputer);
    Console.WriteLine("Index of last computer book in the second half of
the collection: {0}", ndx);
}
```

```
// Populates the list with sample data.
private static void FillList()
{
    // Create XML elements from a source file.
    XElement xTree = XElement.Load(@"c:\temp\books.xml");

    // Create an enumerable collection of the elements.
    IEnumerable< XElement > elements = xTree.Elements();

    // Evaluate each element and set set values in the book object.
    foreach ( XElement el in elements )
    {
        Book book = new Book();
        book.ID = el.Attribute("id").Value;
        IEnumerable< XElement > props = el.Elements();
        foreach ( XElement p in props )
        {

            if ( p.Name.ToString().ToLower() == "author" )
            {
                book.Author = p.Value;
            }
            else if ( p.Name.ToString().ToLower() == "title" )
            {
                book.Title = p.Value;
            }
            else if ( p.Name.ToString().ToLower() == "genre" )
            {
                book.Genre = p.Value;
            }
            else if ( p.Name.ToString().ToLower() == "price" )
            {
                book.Price = Convert.ToDouble(p.Value);
            }
            else if ( p.Name.ToString().ToLower() == "publish_date" )
            {
                book.Publish_date = Convert.ToDateTime(p.Value);
            }
            else if ( p.Name.ToString().ToLower() == "description" )
            {
                book.Description = p.Value;
            }
        }

        Books.Add(book);
    }

    DisplayResults(Books, "All books:");
}

// Explicit predicate delegate.
private static bool FindComputer( Book bk )
{
```

```

        if (bk.Genre == "Computer")
    {
        return true;
    }
else
{
    return false;
}
}

private static void DisplayResult(Book result, string title)
{
    Console.WriteLine();
    Console.WriteLine(title);
    Console.WriteLine("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", result.ID,
        result.Author, result.Title, result.Genre, result.Price,
        result.Publish_date.ToShortDateString());
    Console.WriteLine();
}

private static void DisplayResults(List<Book> results, string title)
{
    Console.WriteLine();
    Console.WriteLine(title);
    foreach (Book b in results)
    {

        Console.Write("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", b.ID,
            b.Author, b.Title, b.Genre, b.Price,
            b.Publish_date.ToShortDateString());
    }
    Console.WriteLine();
}
}

public class Book
{
    public string ID { get; set; }
    public string Author { get; set; }
    public string Title { get; set; }
    public string Genre { get; set; }
    public double Price { get; set; }
    public DateTime Publish_date { get; set; }
    public string Description { get; set; }
}
}

```

Remarks

The `Predicate<T>` is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current `List<T>` are

individually passed to the [Predicate<T>](#) delegate, moving forward in the [List<T>](#), starting with the first element and ending with the last element. Processing is stopped when a match is found.

Important

When searching a list containing value types, make sure the default value for the type does not satisfy the search predicate. Otherwise, there is no way to distinguish between a default value indicating that no match was found and a list element that happens to have the default value for the type. If the default value satisfies the search predicate, use the [FindIndex](#) method instead.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Exists\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindIndex](#)
- [FindLastIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

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

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Retrieves all the elements that match the conditions defined by the specified predicate.

C#

```
public System.Collections.Generic.List<T> FindAll(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the elements to search for.

Returns

[List<T>](#)

A [List<T>](#) containing all the elements that match the conditions defined by the specified predicate, if found; otherwise, an empty [List<T>](#).

Exceptions

[ArgumentNullException](#)

`match` is `null`.

Examples

The following example demonstrates the find methods for the [List<T>](#) class. The example for the [List<T>](#) class contains `book` objects, of class `Book`, using the data from the [Sample XML File: Books \(LINQ to XML\)](#). The `FillList` method in the example uses [LINQ to XML](#) to parse the values from the XML to property values of the `book` objects.

The following table describes the examples provided for the find methods.

 Expand table

Method	Example
Find(Predicate<T>)	Finds a book by an ID using the <code>IDToFind</code> predicate delegate. C# example uses an anonymous delegate.
FindAll(Predicate<T>)	Find all books that whose <code>Genre</code> property is "Computer" using the <code>FindComputer</code> predicate delegate.
FindLast(Predicate<T>)	Finds the last book in the collection that has a publish date before 2001, using the <code>PubBefore2001</code> predicate delegate. C# example uses an anonymous delegate.
FindIndex(Predicate<T>)	Finds the index of first computer book using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Predicate<T>)	Finds the index of the last computer book using the <code>FindComputer</code> predicate delegate.
FindIndex(Int32, Int32, Predicate<T>)	Finds the index of first computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Int32, Int32, Predicate<T>)	Finds the index of last computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace Find
{
    class Program
    {
        private static string IDtoFind = "bk109";

        private static List<Book> Books = new List<Book>();
        public static void Main(string[] args)
        {
            FillList();

            // Find a book by its ID.
            Book result = Books.Find(
                delegate(Book bk)
                {
                    return bk.ID == IDtoFind;
                }
            );
            if (result != null)

```

```
        {
            DisplayResult(result, "Find by ID: " + IDtoFind);
        }
    else
    {
        Console.WriteLine("\nNot found: {0}", IDtoFind);
    }

    // Find last book in collection published before 2001.
    result = Books.FindLast(
        delegate(Book bk)
    {
        DateTime year2001 = new DateTime(2001,01,01);
        return bk.Publish_date < year2001;
    });
    if (result != null)
    {
        DisplayResult(result, "Last book in collection published before
2001:");
    }
    else
    {
        Console.WriteLine("\nNot found: {0}", IDtoFind);
    }

    // Find all computer books.
    List<Book> results = Books.FindAll(FindComputer);
    if (results.Count != 0)
    {
        DisplayResults(results, "All computer:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find all books under $10.00.
    results = Books.FindAll(
        delegate(Book bk)
    {
        return bk.Price < 10.00;
    });
    if (results.Count != 0)
    {
        DisplayResults(results, "Books under $10:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find index values.
    Console.WriteLine();
    int ndx = Books.FindIndex(FindComputer);
```

```

Console.WriteLine("Index of first computer book: {0}", ndx);
ndx = Books.FindLastIndex(FindComputer);
Console.WriteLine("Index of last computer book: {0}", ndx);

int mid = Books.Count / 2;
ndx = Books.FindIndex(mid, mid, FindComputer);
Console.WriteLine("Index of first computer book in the second half of
the collection: {0}", ndx);

ndx = Books.FindLastIndex(Books.Count - 1, mid, FindComputer);
Console.WriteLine("Index of last computer book in the second half of
the collection: {0}", ndx);
}

// Populates the list with sample data.
private static void FillList()
{

    // Create XML elements from a source file.
    XElement xTree = XElement.Load(@"c:\temp\books.xml");

    // Create an enumerable collection of the elements.
    IEnumerable<XElement> elements = xTree.Elements();

    // Evaluate each element and set set values in the book object.
    foreach (XElement el in elements)
    {
        Book book = new Book();
        book.ID = el.Attribute("id").Value;
        IEnumerable<XElement> props = el.Elements();
        foreach (XElement p in props)
        {

            if (p.Name.ToString().ToLower() == "author")
            {
                book.Author = p.Value;
            }
            else if (p.Name.ToString().ToLower() == "title")
            {
                book.Title = p.Value;
            }
            else if (p.Name.ToString().ToLower() == "genre")
            {
                book.Genre = p.Value;
            }
            else if (p.Name.ToString().ToLower() == "price")
            {
                book.Price = Convert.ToDouble(p.Value);
            }
            else if (p.Name.ToString().ToLower() == "publish_date")
            {
                book.Publish_date = Convert.ToDateTime(p.Value);
            }
            else if (p.Name.ToString().ToLower() == "description")
            {

```

```
        book.Description = p.Value;
    }
}

Books.Add(book);
}

DisplayResults(Books, "All books:");
}

// Explicit predicate delegate.
private static bool FindComputer(Book bk)
{
    if (bk.Genre == "Computer")
    {
        return true;
    }
    else
    {
        return false;
    }
}

private static void DisplayResult(Book result, string title)
{
    Console.WriteLine();
    Console.WriteLine(title);
    Console.WriteLine("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", result.ID,
                    result.Author, result.Title, result.Genre, result.Price,
                    result.Publish_date.ToString("MM/dd/yyyy"));
    Console.WriteLine();
}

private static void DisplayResults(List<Book> results, string title)
{
    Console.WriteLine();
    Console.WriteLine(title);
    foreach (Book b in results)
    {
        Console.Write("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", b.ID,
                     b.Author, b.Title, b.Genre, b.Price,
                     b.Publish_date.ToString("MM/dd/yyyy"));
    }
    Console.WriteLine();
}
}

public class Book
{
    public string ID { get; set; }
    public string Author { get; set; }
    public string Title { get; set; }
    public string Genre { get; set; }
```

```
        public double Price { get; set; }
        public DateTime Publish_date { get; set; }
        public string Description { get; set; }
    }
}
```

Remarks

The [Predicate<T>](#) is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current [List<T>](#) are individually passed to the [Predicate<T>](#) delegate, and the elements that match the conditions are saved in the returned [List<T>](#).

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Exists\(Predicate<T>\)](#)
- [Find\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindIndex](#)
- [FindLastIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

List<T>.FindIndex Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches for an element that matches the conditions defined by a specified predicate, and returns the zero-based index of the first occurrence within the [List<T>](#) or a portion of it. This method returns -1 if an item that matches the conditions is not found.

Overloads

 [Expand table](#)

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(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> .
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(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.

C#

```
public int FindIndex(int startIndex, int count, Predicate<T> match);
```

Parameters

startIndex [Int32](#)

The zero-based starting index of the search.

count [Int32](#)

The number of elements in the section to search.

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the element to search for.

Returns

[Int32](#)

The zero-based index of the first occurrence of an element that matches the conditions defined by `match`, if found; otherwise, -1.

Exceptions

[ArgumentNullException](#)

`match` is `null`.

[ArgumentOutOfRangeException](#)

`startIndex` is outside the range of valid indexes for the [List<T>](#).

-or-

`count` is less than 0.

-or-

`startIndex` and `count` do not specify a valid section in the [List<T>](#).

Examples

The following example defines an `Employee` class with two fields, `Name` and `Id`. It also defines an `EmployeeSearch` class with a single method, `StartsWith`, that indicates whether the `Employee.Name` field starts with a specified substring that is supplied to the `EmployeeSearch` class constructor. Note the signature of this method

C#

```
public bool StartsWith(Employee e)
```

corresponds to the signature of the delegate that can be passed to the `FindIndex` method. The example instantiates a `List<Employee>` object, adds a number of `Employee` objects to it, and then calls the `FindIndex(Int32, Int32, Predicate<T>)` method twice to search the entire collection (that is, the members from index 0 to index `Count` - 1). The first time, it searches for the first `Employee` object whose `Name` field begins with "J"; the second time, it searches for the first `Employee` object whose `Name` field begins with "Ju".

C#

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

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o )
    {
        Employee e = o as Employee;
        if (e == null)
            throw new ArgumentException("o is not an Employee object.");

        return Name.CompareTo(e.Name);
    }
}

public class EmployeeSearch
{
    String _s;

    public EmployeeSearch(String s)
    {
        _s = s;
    }

    public bool StartsWith(Employee e)
    {
        return e.Name.StartsWith(_s,
StringComparison.InvariantCultureIgnoreCase);
    }
}

public class Example
{
    public static void Main()
    {
        var employees = new List<Employee>();
        employees.AddRange( new Employee[] { new Employee { Name = "Frank", Id =
2 },
                                         new Employee { Name = "Jill", Id = 3
},
                                         new Employee { Name = "Bob", Id = 4
},
                                         new Employee { Name = "Sue", Id = 5
}
} );
    }
}
```

```

        new Employee { Name = "Dave", Id = 5
    },
        new Employee { Name = "Jack", Id = 8
    },
        new Employee { Name = "Judith", Id =
12 },
        new Employee { Name = "Robert", Id =
14 },
        new Employee { Name = "Adam", Id = 1
} } );
employees.Sort();

var es = new EmployeeSearch("J");
Console.WriteLine("'J' starts at index {0}",
                employees.FindIndex(0, employees.Count - 1,
es.StartsWith));

es = new EmployeeSearch("Ju");
Console.WriteLine("'Ju' starts at index {0}",
                employees.FindIndex(0, employees.Count -
1,es.StartsWith));
}
}
// The example displays the following output:
//      'J' starts at index 3
//      'Ju' starts at index 5

```

Remarks

The `List<T>` is searched forward starting at `startIndex` and ending at `startIndex` plus `count` minus 1, if `count` is greater than 0.

The `Predicate<T>` is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current `List<T>` are individually passed to the `Predicate<T>` delegate. The delegate has the signature:

C#

```
public bool methodName(T obj)
```

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `count`.

See also

- [Exists\(Predicate<T>\)](#)
- [Find\(Predicate<T>\)](#)

- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindLastIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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>](#).

C#

```
public int FindIndex(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the element to search for.

Returns

[Int32](#)

The zero-based index of the first occurrence of an element that matches the conditions defined by **match**, if found; otherwise, -1.

Exceptions

ArgumentNullException

match is null.

Examples

The following example defines an `Employee` class with two fields, `Name` and `Id`. It also defines an `EmployeeSearch` class with a single method, `StartsWith`, that indicates whether the `Employee.Name` field starts with a specified substring that is supplied to the `EmployeeSearch` class constructor. Note the signature of this method

C#

```
public bool StartsWith(Employee e)
```

corresponds to the signature of the delegate that can be passed to the `FindIndex` method. The example instantiates a `List<Employee>` object, adds a number of `Employee` objects to it, and then calls the `FindIndex(Int32, Int32, Predicate<T>)` method twice to search the entire collection, the first time for the first `Employee` object whose `Name` field begins with "J", and the second time for the first `Employee` object whose `Name` field begins with "Ju".

C#

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

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        Employee e = o as Employee;
        if (e == null)
            throw new ArgumentException("o is not an Employee object.");

        return Name.CompareTo(e.Name);
    }
}

public class EmployeeSearch
{
    String _s;

    public EmployeeSearch(String s)
    {
```

```

        _s = s;
    }

    public bool StartsWith(Employee e)
    {
        return e.Name.StartsWith(_s,
StringComparison.InvariantCultureIgnoreCase);
    }
}

public class Example
{
    public static void Main()
    {
        var employees = new List<Employee>();
        employees.AddRange( new Employee[] { new Employee { Name = "Frank", Id =
2 },
                                            new Employee { Name = "Jill", Id = 3
},
                                            new Employee { Name = "Dave", Id = 5
},
                                            new Employee { Name = "Jack", Id = 8
},
                                            new Employee { Name = "Judith", Id =
12 },
                                            new Employee { Name = "Robert", Id =
14 },
                                            new Employee { Name = "Adam", Id = 1
} } );
        employees.Sort();

        var es = new EmployeeSearch("J");
        Console.WriteLine("'J' starts at index {0}",
                        employees.FindIndex(es.StartsWith()));

        es = new EmployeeSearch("Ju");
        Console.WriteLine("'Ju' starts at index {0}",
                        employees.FindIndex(es.StartsWith()));
    }
}
// The example displays the following output:
//      'J' starts at index 3
//      'Ju' starts at index 5

```

Remarks

The `List<T>` is searched forward starting at the first element and ending at the last element.

The `Predicate<T>` is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current `List<T>` are individually passed to the `Predicate<T>` delegate. The delegate has the signature:

C#

```
public bool methodName(T obj)
```

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

See also

- [Exists\(Predicate<T>\)](#)
- [Find\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindLastIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int FindIndex(int startIndex, Predicate<T> match);
```

Parameters

startIndex `Int32`

The zero-based starting index of the search.

match `Predicate<T>`

The `Predicate<T>` delegate that defines the conditions of the element to search for.

Returns

`Int32`

The zero-based index of the first occurrence of an element that matches the conditions defined by `match`, if found; otherwise, -1.

Exceptions

`ArgumentNullException`

`match` is `null`.

`ArgumentOutOfRangeException`

`startIndex` is outside the range of valid indexes for the `List<T>`.

Examples

The following example defines an `Employee` class with two fields, `Name` and `Id`. It also defines an `EmployeeSearch` class with a single method, `StartsWith`, that indicates whether the `Employee.Name` field starts with a specified substring that is supplied to the `EmployeeSearch` class constructor. Note the signature of this method

C#

```
public bool StartsWith(Employee e)
```

corresponds to the signature of the delegate that can be passed to the `FindIndex` method. The example instantiates a `List<Employee>` object, adds a number of `Employee` objects to it, and then calls the `FindIndex(Int32, Int32, Predicate<T>)` method twice to search the collection starting with its fifth member (that is, the member at index 4). The first time, it searches for the first `Employee` object whose `Name` field begins with "J"; the second time, it searches for the first `Employee` object whose `Name` field begins with "Ju".

C#

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

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o )
    {
        Employee e = o as Employee;
        if (e == null)
            throw new ArgumentException("o is not an Employee object.");

        return Name.CompareTo(e.Name);
    }
}

public class EmployeeSearch
{
    String _s;

    public EmployeeSearch(String s)
    {
        _s = s;
    }

    public bool StartsWith(Employee e)
    {
        return e.Name.StartsWith(_s,
StringComparison.InvariantCultureIgnoreCase);
    }
}

public class Example
{
    public static void Main()
    {
        var employees = new List<Employee>();
        employees.AddRange( new Employee[] { new Employee { Name = "Frank", Id =
2 },
                                            new Employee { Name = "Jill", Id = 3 },
                                            new Employee { Name = "Dave", Id = 5 },
                                            new Employee { Name = "Jack", Id = 8 },
                                            new Employee { Name = "Judith", Id =
12 },
                                            new Employee { Name = "Robert", Id =
14 },
                                            new Employee { Name = "Adam", Id = 1
} } );
        employees.Sort();
```

```

        var es = new EmployeeSearch("J");
        int index = employees.FindIndex(4, es.StartsWith);
        Console.WriteLine("Starting index of 'J': {0}",
                           index >= 0 ? index.ToString() : "Not found");

        es = new EmployeeSearch("Ju");
        index = employees.FindIndex(4, es.StartsWith);
        Console.WriteLine("Starting index of 'Ju': {0}",
                           index >= 0 ? index.ToString() : "Not found");
    }
}

// The example displays the following output:
//      'J' starts at index 4
//      'Ju' starts at index 5

```

Remarks

The `List<T>` is searched forward starting at `startIndex` and ending at the last element.

The `Predicate<T>` is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current `List<T>` are individually passed to the `Predicate<T>` delegate. The delegate has the signature:

C#

```
public bool methodName(T obj)
```

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is the number of elements from `startIndex` to the end of the `List<T>`.

See also

- [Exists\(Predicate<T>\)](#)
- [Find\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindLastIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches for an element that matches the conditions defined by the specified predicate, and returns the last occurrence within the entire [List<T>](#).

C#

```
public T? FindLast(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the element to search for.

Returns

T

The last element that matches the conditions defined by the specified predicate, if found; otherwise, the default value for type [T](#).

Exceptions

[ArgumentNullException](#)

`match` is `null`.

Examples

The following example demonstrates the find methods for the [List<T>](#) class. The example for the [List<T>](#) class contains `book` objects, of class `Book`, using the data from the [Sample XML File: Books \(LINQ to XML\)](#). The `FillList` method in the example uses [LINQ to XML](#) to parse the values from the XML to property values of the `book` objects.

The following table describes the examples provided for the find methods.

Method	Example
Find(Predicate<T>)	Finds a book by an ID using the <code>IDToFind</code> predicate delegate. C# example uses an anonymous delegate.
FindAll(Predicate<T>)	Find all books that whose <code>Genre</code> property is "Computer" using the <code>FindComputer</code> predicate delegate.
FindLast(Predicate<T>)	Finds the last book in the collection that has a publish date before 2001, using the <code>PubBefore2001</code> predicate delegate. C# example uses an anonymous delegate.
FindIndex(Predicate<T>)	Finds the index of first computer book using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Predicate<T>)	Finds the index of the last computer book using the <code>FindComputer</code> predicate delegate.
FindIndex(Int32, Int32, Predicate<T>)	Finds the index of first computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Int32, Int32, Predicate<T>)	Finds the index of last computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace Find
{
    class Program
    {
        private static string IDtoFind = "bk109";

        private static List<Book> Books = new List<Book>();
        public static void Main(string[] args)
        {
            FillList();

            // Find a book by its ID.
            Book result = Books.Find(
                delegate(Book bk)
                {
                    return bk.ID == IDtoFind;
                }
            );
        }
    }
}

```

```
    );
    if (result != null)
    {
        DisplayResult(result, "Find by ID: " + IDtoFind);
    }
    else
    {
        Console.WriteLine("\nNot found: {0}", IDtoFind);
    }

    // Find last book in collection published before 2001.
    result = Books.FindLast(
        delegate(Book bk)
    {
        DateTime year2001 = new DateTime(2001,01,01);
        return bk.Publish_date < year2001;
    });
    if (result != null)
    {
        DisplayResult(result, "Last book in collection published before
2001:");
    }
    else
    {
        Console.WriteLine("\nNot found: {0}", IDtoFind);
    }

    // Find all computer books.
    List<Book> results = Books.FindAll(FindComputer);
    if (results.Count != 0)
    {
        DisplayResults(results, "All computer:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find all books under $10.00.
    results = Books.FindAll(
        delegate(Book bk)
    {
        return bk.Price < 10.00;
    });
    if (results.Count != 0)
    {
        DisplayResults(results, "Books under $10:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find index values.
```

```
Console.WriteLine();
int ndx = Books.FindIndex(FindComputer);
Console.WriteLine("Index of first computer book: {0}", ndx);
ndx = Books.FindLastIndex(FindComputer);
Console.WriteLine("Index of last computer book: {0}", ndx);

int mid = Books.Count / 2;
ndx = Books.FindIndex(mid, mid, FindComputer);
Console.WriteLine("Index of first computer book in the second half of
the collection: {0}", ndx);

ndx = Books.FindLastIndex(Books.Count - 1, mid, FindComputer);
Console.WriteLine("Index of last computer book in the second half of
the collection: {0}", ndx);
}

// Populates the list with sample data.
private static void FillList()
{

    // Create XML elements from a source file.
    XElement xTree = XElement.Load(@"c:\temp\books.xml");

    // Create an enumerable collection of the elements.
    IEnumerable<XElement> elements = xTree.Elements();

    // Evaluate each element and set set values in the book object.
    foreach (XElement el in elements)
    {
        Book book = new Book();
        book.ID = el.Attribute("id").Value;
        IEnumerable<XElement> props = el.Elements();
        foreach ( XElement p in props)
        {

            if (p.Name.ToString().ToLower() == "author")
            {
                book.Author = p.Value;
            }
            else if (p.Name.ToString().ToLower() == "title")
            {
                book.Title = p.Value;
            }
            else if (p.Name.ToString().ToLower() == "genre")
            {
                book.Genre = p.Value;
            }
            else if (p.Name.ToString().ToLower() == "price")
            {
                book.Price = Convert.ToDouble(p.Value);
            }
            else if (p.Name.ToString().ToLower() == "publish_date")
            {
                book.Publish_date = Convert.ToDateTime(p.Value);
            }
        }
    }
}
```

```
        else if (p.Name.ToString().ToLower() == "description")
    {
        book.Description = p.Value;
    }
}

Books.Add(book);
}

DisplayResults(Books, "All books:");
}

// Explicit predicate delegate.
private static bool FindComputer(Book bk)
{

    if (bk.Genre == "Computer")
    {
        return true;
    }
    else
    {
        return false;
    }
}

private static void DisplayResult(Book result, string title)
{
    Console.WriteLine();
    Console.WriteLine(title);
    Console.WriteLine("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", result.ID,
        result.Author, result.Title, result.Genre, result.Price,
        result.Publish_date.ToShortDateString());
    Console.WriteLine();
}

private static void DisplayResults(List<Book> results, string title)
{
    Console.WriteLine();
    Console.WriteLine(title);
    foreach (Book b in results)
    {

        Console.Write("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", b.ID,
            b.Author, b.Title, b.Genre, b.Price,
            b.Publish_date.ToShortDateString());
    }
    Console.WriteLine();
}
}

public class Book
{
    public string ID { get; set; }
    public string Author { get; set; }
```

```

        public string Title { get; set; }
        public string Genre { get; set; }
        public double Price { get; set; }
        public DateTime Publish_date { get; set; }
        public string Description { get; set; }
    }
}

```

Remarks

The [Predicate<T>](#) is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current [List<T>](#) are individually passed to the [Predicate<T>](#) delegate, moving backward in the [List<T>](#), starting with the last element and ending with the first element. Processing is stopped when a match is found.

Important

When searching a list containing value types, make sure the default value for the type does not satisfy the search predicate. Otherwise, there is no way to distinguish between a default value indicating that no match was found and a list element that happens to have the default value for the type. If the default value satisfies the search predicate, use the [FindLastIndex](#) method instead.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Exists\(Predicate<T>\)](#)

- [Find\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindIndex](#)
- [FindLastIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

List<T>.FindLastIndex Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches for an element that matches the conditions defined by a specified predicate, and returns the zero-based index of the last occurrence within the [List<T>](#) or a portion of it.

Overloads

 [Expand table](#)

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> .
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(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(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>](#).

C#

```
public int FindLastIndex(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the element to search for.

Returns

Int32

The zero-based index of the last occurrence of an element that matches the conditions defined by `match`, if found; otherwise, -1.

Exceptions

[ArgumentNullException](#)

`match` is `null`.

Examples

The following example demonstrates the find methods for the `List<T>` class. The example for the `List<T>` class contains `book` objects, of class `Book`, using the data from the [Sample XML File: Books \(LINQ to XML\)](#). The `FillList` method in the example uses [LINQ to XML](#) to parse the values from the XML to property values of the `book` objects.

The following table describes the examples provided for the find methods.

[+] [Expand table](#)

Method	Example
Find(Predicate<T>)	Finds a book by an ID using the <code>IDToFind</code> predicate delegate. C# example uses an anonymous delegate.
FindAll(Predicate<T>)	Find all books that whose <code>Genre</code> property is "Computer" using the <code>FindComputer</code> predicate delegate.
FindLast(Predicate<T>)	Finds the last book in the collection that has a publish date before 2001, using the <code>PubBefore2001</code> predicate delegate. C# example uses an anonymous delegate.
FindIndex(Predicate<T>)	Finds the index of first computer book using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Predicate<T>)	Finds the index of the last computer book using the <code>FindComputer</code> predicate delegate.
FindIndex(Int32, Int32, Predicate<T>)	Finds the index of first computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.

Method	Example
<code>FindLastIndex(Int32, Int32, Predicate<T>)</code>	Finds the index of last computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace Find
{
    class Program
    {
        private static string IDtoFind = "bk109";

        private static List<Book> Books = new List<Book>();
        public static void Main(string[] args)
        {
            FillList();

            // Find a book by its ID.
            Book result = Books.Find(
                delegate(Book bk)
                {
                    return bk.ID == IDtoFind;
                }
            );
            if (result != null)
            {
                DisplayResult(result, "Find by ID: " + IDtoFind);
            }
            else
            {
                Console.WriteLine("\nNot found: {0}", IDtoFind);
            }

            // Find last book in collection published before 2001.
            result = Books.FindLast(
                delegate(Book bk)
                {
                    DateTime year2001 = new DateTime(2001, 01, 01);
                    return bk.Publish_date < year2001;
                });
            if (result != null)
            {
                DisplayResult(result, "Last book in collection published before
2001:");
            }
            else
            {

```

```

        Console.WriteLine("\nNot found: {0}", IDtoFind);
    }

    // Find all computer books.
    List<Book> results = Books.FindAll(FindComputer);
    if (results.Count != 0)
    {
        DisplayResults(results, "All computer:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find all books under $10.00.
    results = Books.FindAll(
        delegate(Book bk)
    {
        return bk.Price < 10.00;
    });
    if (results.Count != 0)
    {
        DisplayResults(results, "Books under $10:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find index values.
    Console.WriteLine();
    int ndx = Books.FindIndex(FindComputer);
    Console.WriteLine("Index of first computer book: {0}", ndx);
    ndx = Books.FindLastIndex(FindComputer);
    Console.WriteLine("Index of last computer book: {0}", ndx);

    int mid = Books.Count / 2;
    ndx = Books.FindIndex(mid, mid, FindComputer);
    Console.WriteLine("Index of first computer book in the second half
of the collection: {0}", ndx);

    ndx = Books.FindLastIndex(Books.Count - 1, mid, FindComputer);
    Console.WriteLine("Index of last computer book in the second half
of the collection: {0}", ndx);
}

// Populates the list with sample data.
private static void FillList()
{
    // Create XML elements from a source file.
    XElement xTree = XElement.Load(@"c:\temp\books.xml");

    // Create an enumerable collection of the elements.
}

```

```

IEnumerable< XElement> elements = xTree.Elements();

// Evaluate each element and set set values in the book object.
foreach ( XElement el in elements)
{
    Book book = new Book();
    book.ID = el.Attribute("id").Value;
    IEnumerable< XElement> props = el.Elements();
    foreach ( XElement p in props)
    {

        if (p.Name.ToString().ToLower() == "author")
        {
            book.Author = p.Value;
        }
        else if (p.Name.ToString().ToLower() == "title")
        {
            book.Title = p.Value;
        }
        else if (p.Name.ToString().ToLower() == "genre")
        {
            book.Genre = p.Value;
        }
        else if (p.Name.ToString().ToLower() == "price")
        {
            book.Price = Convert.ToDouble(p.Value);
        }
        else if (p.Name.ToString().ToLower() == "publish_date")
        {
            book.Publish_date = Convert.ToDateTime(p.Value);
        }
        else if (p.Name.ToString().ToLower() == "description")
        {
            book.Description = p.Value;
        }
    }

    Books.Add(book);
}

DisplayResults(Books, "All books:");
}

// Explicit predicate delegate.
private static bool FindComputer(Book bk)
{

    if (bk.Genre == "Computer")
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```

    }

    private static void DisplayResult(Book result, string title)
    {
        Console.WriteLine();
        Console.WriteLine(title);
        Console.WriteLine("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", result.ID,
            result.Author, result.Title, result.Genre, result.Price,
            result.Publish_date.ToShortDateString());
        Console.WriteLine();
    }

    private static void DisplayResults(List<Book> results, string title)
    {
        Console.WriteLine();
        Console.WriteLine(title);
        foreach (Book b in results)
        {

            Console.Write("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", b.ID,
                b.Author, b.Title, b.Genre, b.Price,
                b.Publish_date.ToShortDateString());
        }
        Console.WriteLine();
    }
}

public class Book
{
    public string ID { get; set; }
    public string Author { get; set; }
    public string Title { get; set; }
    public string Genre { get; set; }
    public double Price { get; set; }
    public DateTime Publish_date { get; set; }
    public string Description { get; set; }
}
}

```

Remarks

The `List<T>` is searched backward starting at the last element and ending at the first element.

The `Predicate<T>` is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current `List<T>` are individually passed to the `Predicate<T>` delegate.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `Count`.

See also

- [Exists\(Predicate<T>\)](#)
- [Find\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int FindLastIndex(int startIndex, Predicate<T> match);
```

Parameters

startIndex [Int32](#)

The zero-based starting index of the backward search.

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the element to search for.

Returns

[Int32](#)

The zero-based index of the last occurrence of an element that matches the conditions defined by `match`, if found; otherwise, -1.

Exceptions

[ArgumentNullException](#)

`match` is `null`.

[ArgumentOutOfRangeException](#)

`startIndex` is outside the range of valid indexes for the [List<T>](#).

Remarks

The [List<T>](#) is searched backward starting at `startIndex` and ending at the first element.

The [Predicate<T>](#) is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current [List<T>](#) are individually passed to the [Predicate<T>](#) delegate.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is the number of elements from the beginning of the [List<T>](#) to `startIndex`.

See also

- [Exists\(Predicate<T>\)](#)
- [Find\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int FindLastIndex(int startIndex, int count, Predicate<T> match);
```

Parameters

startIndex [Int32](#)

The zero-based starting index of the backward search.

count [Int32](#)

The number of elements in the section to search.

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the element to search for.

Returns

[Int32](#)

The zero-based index of the last occurrence of an element that matches the conditions defined by `match`, if found; otherwise, -1.

Exceptions

[ArgumentNullException](#)

`match` is `null`.

ArgumentOutOfRangeException

`startIndex` is outside the range of valid indexes for the `List<T>`.

-or-

`count` is less than 0.

-or-

`startIndex` and `count` do not specify a valid section in the `List<T>`.

Examples

The following example demonstrates the find methods for the `List<T>` class. The example for the `List<T>` class contains `book` objects, of class `Book`, using the data from the [Sample XML File: Books \(LINQ to XML\)](#). The `FillList` method in the example uses [LINQ to XML](#) to parse the values from the XML to property values of the `book` objects.

The following table describes the examples provided for the find methods.

 Expand table

Method	Example
Find(Predicate<T>)	Finds a book by an ID using the <code>IDToFind</code> predicate delegate. C# example uses an anonymous delegate.
FindAll(Predicate<T>)	Find all books that whose <code>Genre</code> property is "Computer" using the <code>FindComputer</code> predicate delegate.
FindLast(Predicate<T>)	Finds the last book in the collection that has a publish date before 2001, using the <code>PubBefore2001</code> predicate delegate. C# example uses an anonymous delegate.
FindIndex(Predicate<T>)	Finds the index of first computer book using the <code>FindComputer</code> predicate delegate.
FindLastIndex(Predicate<T>)	Finds the index of the last computer book using the <code>FindComputer</code> predicate delegate.
FindIndex(Int32, Int32, Predicate<T>)	Finds the index of first computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.

Method	Example
<code>FindLastIndex(Int32, Int32, Predicate<T>)</code>	Finds the index of last computer book in the second half of the collection, using the <code>FindComputer</code> predicate delegate.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace Find
{
    class Program
    {
        private static string IDtoFind = "bk109";

        private static List<Book> Books = new List<Book>();
        public static void Main(string[] args)
        {
            FillList();

            // Find a book by its ID.
            Book result = Books.Find(
                delegate(Book bk)
                {
                    return bk.ID == IDtoFind;
                }
            );
            if (result != null)
            {
                DisplayResult(result, "Find by ID: " + IDtoFind);
            }
            else
            {
                Console.WriteLine("\nNot found: {0}", IDtoFind);
            }

            // Find last book in collection published before 2001.
            result = Books.FindLast(
                delegate(Book bk)
                {
                    DateTime year2001 = new DateTime(2001, 01, 01);
                    return bk.Publish_date < year2001;
                });
            if (result != null)
            {
                DisplayResult(result, "Last book in collection published before
2001:");
            }
            else
            {

```

```

        Console.WriteLine("\nNot found: {0}", IDtoFind);
    }

    // Find all computer books.
    List<Book> results = Books.FindAll(FindComputer);
    if (results.Count != 0)
    {
        DisplayResults(results, "All computer:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find all books under $10.00.
    results = Books.FindAll(
        delegate(Book bk)
    {
        return bk.Price < 10.00;
    });
    if (results.Count != 0)
    {
        DisplayResults(results, "Books under $10:");
    }
    else
    {
        Console.WriteLine("\nNo books found.");
    }

    // Find index values.
    Console.WriteLine();
    int ndx = Books.FindIndex(FindComputer);
    Console.WriteLine("Index of first computer book: {0}", ndx);
    ndx = Books.FindLastIndex(FindComputer);
    Console.WriteLine("Index of last computer book: {0}", ndx);

    int mid = Books.Count / 2;
    ndx = Books.FindIndex(mid, mid, FindComputer);
    Console.WriteLine("Index of first computer book in the second half
of the collection: {0}", ndx);

    ndx = Books.FindLastIndex(Books.Count - 1, mid, FindComputer);
    Console.WriteLine("Index of last computer book in the second half
of the collection: {0}", ndx);
}

// Populates the list with sample data.
private static void FillList()
{
    // Create XML elements from a source file.
    XElement xTree = XElement.Load(@"c:\temp\books.xml");

    // Create an enumerable collection of the elements.
}

```

```

IEnumerable< XElement> elements = xTree.Elements();

// Evaluate each element and set set values in the book object.
foreach ( XElement el in elements)
{
    Book book = new Book();
    book.ID = el.Attribute("id").Value;
    IEnumerable< XElement> props = el.Elements();
    foreach ( XElement p in props)
    {

        if (p.Name.ToString().ToLower() == "author")
        {
            book.Author = p.Value;
        }
        else if (p.Name.ToString().ToLower() == "title")
        {
            book.Title = p.Value;
        }
        else if (p.Name.ToString().ToLower() == "genre")
        {
            book.Genre = p.Value;
        }
        else if (p.Name.ToString().ToLower() == "price")
        {
            book.Price = Convert.ToDouble(p.Value);
        }
        else if (p.Name.ToString().ToLower() == "publish_date")
        {
            book.Publish_date = Convert.ToDateTime(p.Value);
        }
        else if (p.Name.ToString().ToLower() == "description")
        {
            book.Description = p.Value;
        }
    }

    Books.Add(book);
}

DisplayResults(Books, "All books:");
}

// Explicit predicate delegate.
private static bool FindComputer(Book bk)
{

    if (bk.Genre == "Computer")
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```

    }

    private static void DisplayResult(Book result, string title)
    {
        Console.WriteLine();
        Console.WriteLine(title);
        Console.WriteLine("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", result.ID,
            result.Author, result.Title, result.Genre, result.Price,
            result.Publish_date.ToShortDateString());
        Console.WriteLine();
    }

    private static void DisplayResults(List<Book> results, string title)
    {
        Console.WriteLine();
        Console.WriteLine(title);
        foreach (Book b in results)
        {

            Console.Write("\n{0}\t{1}\t{2}\t{3}\t{4}\t{5}", b.ID,
                b.Author, b.Title, b.Genre, b.Price,
                b.Publish_date.ToShortDateString());
        }
        Console.WriteLine();
    }
}

public class Book
{
    public string ID { get; set; }
    public string Author { get; set; }
    public string Title { get; set; }
    public string Genre { get; set; }
    public double Price { get; set; }
    public DateTime Publish_date { get; set; }
    public string Description { get; set; }
}
}

```

Remarks

The `List<T>` is searched backward starting at `startIndex` and ending at `startIndex` minus `count` plus 1, if `count` is greater than 0.

The `Predicate<T>` is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current `List<T>` are individually passed to the `Predicate<T>` delegate.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `count`.

See also

- [Exists\(Predicate<T>\)](#)
- [Find\(Predicate<T>\)](#)
- [FindLast\(Predicate<T>\)](#)
- [FindAll\(Predicate<T>\)](#)
- [FindIndex](#)
- [BinarySearch](#)
- [IndexOf](#)
- [LastIndexOf](#)
- [Predicate<T>](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.ForEach(Action<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Performs the specified action on each element of the [List<T>](#).

C#

```
public void ForEach(Action<T> action);
```

Parameters

action [Action<T>](#)

The [Action<T>](#) delegate to perform on each element of the [List<T>](#).

Exceptions

[ArgumentNullException](#)

`action` is `null`.

[InvalidOperationException](#)

An element in the collection has been modified.

Examples

The following example demonstrates the use of the [Action<T>](#) delegate to print the contents of a [List<T>](#) object. In this example the `Print` method is used to display the contents of the list to the console.

(!) Note

In addition to displaying the contents using the `Print` method, the C# example demonstrates the use of [anonymous methods](#) to display the results to the console.

C#

```

List<string> names = new List<string>();
names.Add("Bruce");
names.Add("Alfred");
names.Add("Tim");
names.Add("Richard");

// Display the contents of the list using the Print method.
names.ForEach(Print);

// The following demonstrates the anonymous method feature of C#
// to display the contents of the list to the console.
names.ForEach(delegate(string name)
{
    Console.WriteLine(name);
});

void Print(string s)
{
    Console.WriteLine(s);
}

/* This code will produce output similar to the following:
* Bruce
* Alfred
* Tim
* Richard
* Bruce
* Alfred
* Tim
* Richard
*/

```

Remarks

The [Action<T>](#) is a delegate to a method that performs an action on the object passed to it. The elements of the current [List<T>](#) are individually passed to the [Action<T>](#) delegate.

This method is an $O(n)$ operation, where n is [Count](#).

Modifying the underlying collection in the body of the [Action<T>](#) delegate is not supported and causes undefined behavior.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Action<T>](#)

List<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [List<T>](#).

C#

```
public System.Collections.Generic.List<T>.Enumerator GetEnumerator();
```

Returns

[List<T>.Enumerator](#)

A [List<T>.Enumerator](#) for the [List<T>](#).

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [List<T>.Enumerator](#)
- [IEnumerator<T>](#)

List<T>.GetRange(Int32, Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Creates a shallow copy of a range of elements in the source [List<T>](#).

C#

```
public System.Collections.Generic.List<T> GetRange(int index, int count);
```

Parameters

index [Int32](#)

The zero-based [List<T>](#) index at which the range starts.

count [Int32](#)

The number of elements in the range.

Returns

[List<T>](#)

A shallow copy of a range of elements in the source [List<T>](#).

Exceptions

[ArgumentOutOfRangeException](#)

index is less than 0.

-or-

count is less than 0.

[ArgumentException](#)

index and **count** do not denote a valid range of elements in the [List<T>](#).

Examples

The following example demonstrates the [GetRange](#) method and other methods of the `List<T>` class that act on ranges. At the end of the example, the [GetRange](#) method is used to get three items from the list, beginning with index location 2. The [ToArray](#) method is called on the resulting `List<T>`, creating an array of three elements. The elements of the array are displayed.

C#

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

public class Example
{
    public static void Main()
    {
        string[] input = { "Brachiosaurus",
                           "Amargasaurus",
                           "Mamenchisaurus" };

        List<string> dinosaurs = new List<string>(input);

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nAddRange(dinosaurs)");
        dinosaurs.AddRange(dinosaurs);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nRemoveRange(2, 2)");
        dinosaurs.RemoveRange(2, 2);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        input = new string[] { "Tyrannosaurus",
                              "Deinonychus",
                              "Velociraptor" };

        Console.WriteLine("\nInsertRange(3, input)");
        dinosaurs.InsertRange(3, input);
```

```
Console.WriteLine();
foreach( string dinosaur in dinosaurs )
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\noutput = dinosaurs.GetRange(2, 3).ToArray()");
string[] output = dinosaurs.GetRange(2, 3).ToArray();

Console.WriteLine();
foreach( string dinosaur in output )
{
    Console.WriteLine(dinosaur);
}
}

/* This code example produces the following output:
```

Capacity: 3

Brachiosaurus
Amargasaurus
Mamenchisaurus

AddRange(dinosaurs)

Brachiosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Amargasaurus
Mamenchisaurus

RemoveRange(2, 2)

Brachiosaurus
Amargasaurus
Amargasaurus
Mamenchisaurus

InsertRange(3, input)

Brachiosaurus
Amargasaurus
Amargasaurus
Tyrannosaurus
Deinonychus
Velociraptor
Mamenchisaurus

output = dinosaurs.GetRange(2, 3).ToArray()

Amargasaurus
Tyrannosaurus

Remarks

A shallow copy of a collection of reference types, or a subset of that collection, contains only the references to the elements of the collection. The objects themselves are not copied. The references in the new list point to the same objects as the references in the original list.

A shallow copy of a collection of value types, or a subset of that collection, contains the elements of the collection. However, if the elements of the collection contain references to other objects, those objects are not copied. The references in the elements of the new collection point to the same objects as the references in the elements of the original collection.

In contrast, a deep copy of a collection copies the elements and everything directly or indirectly referenced by the elements.

This method is an $O(n)$ operation, where n is `count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [RemoveRange\(Int32, Int32\)](#)
- [AddRange\(IEnumerable<T>\)](#)
- [InsertRange\(Int32, IEnumerable<T>\)](#)

List<T>.IndexOf Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns the zero-based index of the first occurrence of a value in the [List<T>](#) or in a portion of it.

Overloads

 [Expand table](#)

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.
IndexOf(T)	Searches for the specified object and returns the zero-based index of the first occurrence within the entire List<T> .

Examples

The following example demonstrates all three overloads of the [IndexOf](#) method. A [List<T>](#) of strings is created, with one entry that appears twice, at index location 0 and index location 5. The [IndexOf\(T\)](#) method overload searches the list from the beginning, and finds the first occurrence of the string. The [IndexOf\(T, Int32\)](#) method overload is used to search the list beginning with index location 3 and continuing to the end of the list, and finds the second occurrence of the string. Finally, the [IndexOf\(T, Int32, Int32\)](#) method overload is used to search a range of two entries, beginning at index location two; it returns -1 because there are no instances of the search string in that range.

C#

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

public class Example
{
```

```

public static void Main()
{
    List<string> dinosaurs = new List<string>();

    dinosaurs.Add("Tyrannosaurus");
    dinosaurs.Add("Amargasaurus");
    dinosaurs.Add("Mamenchisaurus");
    dinosaurs.Add("Brachiosaurus");
    dinosaurs.Add("Deinonychus");
    dinosaurs.Add("Tyrannosaurus");
    dinosaurs.Add("Compsognathus");

    Console.WriteLine();
    foreach(string dinosaur in dinosaurs)
    {
        Console.WriteLine(dinosaur);
    }

    Console.WriteLine("\nIndexOf(\"Tyrannosaurus\"): {0}",
        dinosaurs.IndexOf("Tyrannosaurus"));

    Console.WriteLine("\nIndexOf(\"Tyrannosaurus\", 3): {0}",
        dinosaurs.IndexOf("Tyrannosaurus", 3));

    Console.WriteLine("\nIndexOf(\"Tyrannosaurus\", 2, 2): {0}",
        dinosaurs.IndexOf("Tyrannosaurus", 2, 2));
}
}

```

/ This code example produces the following output:*

```

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Deinonychus
Tyrannosaurus
Compsognathus

```

```
IndexOf("Tyrannosaurus"): 0
```

```
IndexOf("Tyrannosaurus", 3): 5
```

```
IndexOf("Tyrannosaurus", 2, 2): -1
*/
```

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.

C#

```
public int IndexOf(T item, int index);
```

Parameters

item `T`

The object to locate in the `List<T>`. The value can be `null` for reference types.

index `Int32`

The zero-based starting index of the search. 0 (zero) is valid in an empty list.

Returns

`Int32`

The zero-based index of the first occurrence of `item` within the range of elements in the `List<T>` that extends from `index` to the last element, if found; otherwise, -1.

Exceptions

`ArgumentOutOfRangeException`

`index` is outside the range of valid indexes for the `List<T>`.

Remarks

The `List<T>` is searched forward starting at `index` and ending at the last element.

This method determines equality using the default equality comparer `EqualityComparer<T>.Default` for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is the number of elements from `index` to the end of the `List<T>`.

See also

- [LastIndexOf\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int IndexOf(T item, int index, int count);
```

Parameters

item [T](#)

The object to locate in the [List<T>](#). The value can be `null` for reference types.

index [Int32](#)

The zero-based starting index of the search. 0 (zero) is valid in an empty list.

count [Int32](#)

The number of elements in the section to search.

Returns

[Int32](#)

The zero-based index of the first occurrence of `item` within the range of elements in the [List<T>](#) that starts at `index` and contains `count` number of elements, if found; otherwise, -1.

Exceptions

ArgumentOutOfRangeException

`index` is outside the range of valid indexes for the [List<T>](#).

-or-

`count` is less than 0.

-or-

`index` and `count` do not specify a valid section in the [List<T>](#).

Remarks

The [List<T>](#) is searched forward starting at `index` and ending at `index` plus `count` minus 1, if `count` is greater than 0.

This method determines equality using the default equality comparer [EqualityComparer<T>.Default](#) for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `count`.

See also

- [LastIndexOf\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

IndexOf(T)

Searches for the specified object and returns the zero-based index of the first occurrence within the entire [List<T>](#).

C#

```
public int IndexOf(T item);
```

Parameters

item `T`

The object to locate in the [List<T>](#). The value can be `null` for reference types.

Returns

`Int32`

The zero-based index of the first occurrence of `item` within the entire [List<T>](#), if found; otherwise, -1.

Implements

[IndexOf\(T\)](#)

Remarks

The [List<T>](#) is searched forward starting at the first element and ending at the last element.

This method determines equality using the default equality comparer [EqualityComparer<T>.Default](#) for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

See also

- [LastIndexOf\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.Insert(Int32, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Inserts an element into the [List<T>](#) at the specified index.

C#

```
public void Insert(int index, T item);
```

Parameters

index [Int32](#)

The zero-based index at which **item** should be inserted.

item [T](#)

The object to insert. The value can be [null](#) for reference types.

Implements

[Insert\(Int32, T\)](#)

Exceptions

[ArgumentOutOfRangeException](#)

index is less than 0.

-or-

index is greater than [Count](#).

Examples

The following example demonstrates how to add, remove, and insert a simple business object in a [List<T>](#).

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify the type of part
// but the part name can change.
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public override int GetHashCode()
    {
        return PartId;
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        // Write out the parts in the list. This will call the overridden ToString
        // method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }
    }
}
```

```

}

// Check the list for part #1734. This calls the IEquatable.Equals method
// of the Part class, which checks the PartId for equality.
Console.WriteLine("\nContains(\"1734\"): {0}",
parts.Contains(new Part { PartId = 1734, PartName = "" }));

// Insert a new item at position 2.
Console.WriteLine("\nInsert(2, \"1834\")");
parts.Insert(2, new Part() { PartName = "brake lever", PartId = 1834 });

//Console.WriteLine();
foreach (Part aPart in parts)
{
    Console.WriteLine(aPart);
}

Console.WriteLine("\nParts[3]: {0}", parts[3]);

Console.WriteLine("\nRemove(\"1534\")");

// This will remove part 1534 even though the PartName is different,
// because the Equals method only checks PartId for equality.
parts.Remove(new Part() { PartId = 1534, PartName = "cogs" });

Console.WriteLine();
foreach (Part aPart in parts)
{
    Console.WriteLine(aPart);
}
Console.WriteLine("\nRemoveAt(3)");
// This will remove the part at index 3.
parts.RemoveAt(3);

Console.WriteLine();
foreach (Part aPart in parts)
{
    Console.WriteLine(aPart);
}

/*
   ID: 1234  Name: crank arm
   ID: 1334  Name: chain ring
   ID: 1434  Name: regular seat
   ID: 1444  Name: banana seat
   ID: 1534  Name: cassette
   ID: 1634  Name: shift lever

   Contains("1734"): False

   Insert(2, "1834")
   ID: 1234  Name: crank arm
   ID: 1334  Name: chain ring
   ID: 1834  Name: brake lever

```

```

        ID: 1434  Name: regular seat
        ID: 1444  Name: banana seat
        ID: 1534  Name: cassette
        ID: 1634  Name: shift lever

    Parts[3]: ID: 1434  Name: regular seat

    Remove("1534")

        ID: 1234  Name: crank arm
        ID: 1334  Name: chain ring
        ID: 1834  Name: brake lever
        ID: 1434  Name: regular seat
        ID: 1444  Name: banana seat
        ID: 1634  Name: shift lever

    RemoveAt(3)

        ID: 1234  Name: crank arm
        ID: 1334  Name: chain ring
        ID: 1834  Name: brake lever
        ID: 1444  Name: banana seat
        ID: 1634  Name: shift lever

    */
}
}

```

The following example demonstrates the [Insert](#) method, along with various other properties and methods of the [List<T>](#) generic class. After the list is created, elements are added. The [Insert](#) method is used to insert an item into the middle of the list. The item inserted is a duplicate, which is later removed using the [Remove](#) method.

```
C#
List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
```

```
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\"));"
dinosaurs.Insert(2, "Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

// Shows accessing the list using the Item property.
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

Console.WriteLine("\nRemove(\"Compsognathus\"));"
dinosaurs.Remove("Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

/* This code example produces the following output:
```

```
Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8
Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
```

```

Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosaurs[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

TrimExcess()
Capacity: 5
Count: 5

Clear()
Capacity: 5
Count: 0
*/

```

Remarks

`List<T>` accepts `null` as a valid value for reference types and allows duplicate elements.

If `Count` already equals `Capacity`, the capacity of the `List<T>` is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If `index` is equal to `Count`, `item` is added to the end of `List<T>`.

This method is an $O(n)$ operation, where n is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [InsertRange\(Int32, IEnumerable<T>\)](#)
- [Add\(T\)](#)
- [Remove\(T\)](#)

List<T>.InsertRange(Int32, IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Inserts the elements of a collection into the [List<T>](#) at the specified index.

C#

```
public void InsertRange(int index, System.Collections.Generic.IEnumerable<T>
collection);
```

Parameters

index Int32

The zero-based index at which the new elements should be inserted.

collection IEnumerable<T>

The collection whose elements should be inserted into the [List<T>](#). The collection itself cannot be `null`, but it can contain elements that are `null`, if type `T` is a reference type.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

[ArgumentOutOfRangeException](#)

`index` is less than 0.

-or-

`index` is greater than `Count`.

Examples

The following example demonstrates [InsertRange](#) method and various other methods of the [List<T>](#) class that act on ranges. After the list has been created and populated with the names

of several peaceful plant-eating dinosaurs, the `InsertRange` method is used to insert an array of three ferocious meat-eating dinosaurs into the list, beginning at index location 3.

C#

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

public class Example
{
    public static void Main()
    {
        string[] input = { "Brachiosaurus",
                           "Amargasaurus",
                           "Mamenchisaurus" };

        List<string> dinosaurs = new List<string>(input);

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nAddRange(dinosaurs)");
        dinosaurs.AddRange(dinosaurs);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nRemoveRange(2, 2)");
        dinosaurs.RemoveRange(2, 2);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        input = new string[] { "Tyrannosaurus",
                              "Deinonychus",
                              "Velociraptor" };

        Console.WriteLine("\nInsertRange(3, input)");
        dinosaurs.InsertRange(3, input);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
```

```
        Console.WriteLine(dinosaur);
    }

    Console.WriteLine("\noutput = dinosaurs.GetRange(2, 3).ToArray()");
    string[] output = dinosaurs.GetRange(2, 3).ToArray();

    Console.WriteLine();
    foreach( string dinosaur in output )
    {
        Console.WriteLine(dinosaur);
    }
}

/* This code example produces the following output:

Capacity: 3

Brachiosaurus
Amargasaurus
Mamenchisaurus

AddRange(dinosaurs)

Brachiosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Amargasaurus
Mamenchisaurus

RemoveRange(2, 2)

Brachiosaurus
Amargasaurus
Amargasaurus
Mamenchisaurus

InsertRange(3, input)

Brachiosaurus
Amargasaurus
Amargasaurus
Tyrannosaurus
Deinonychus
Velociraptor
Mamenchisaurus

output = dinosaurs.GetRange(2, 3).ToArray()

Amargasaurus
Tyrannosaurus
Deinonychus
*/
```

Remarks

`List<T>` accepts `null` as a valid value for reference types and allows duplicate elements.

If the new `Count` (the current `Count` plus the size of the collection) will be greater than `Capacity`, the capacity of the `List<T>` is increased by automatically reallocating the internal array to accommodate the new elements, and the existing elements are copied to the new array before the new elements are added.

If `index` is equal to `Count`, the elements are added to the end of `List<T>`.

The order of the elements in the collection is preserved in the `List<T>`.

This method is an $O(n * m)$ operation, where n is the number of elements to be added and m is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Insert\(Int32, T\)](#)
- [AddRange\(IEnumerable<T>\)](#)
- [GetRange\(Int32, Int32\)](#)
- [RemoveRange\(Int32, Int32\)](#)

List<T>.LastIndexOf Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns the zero-based index of the last occurrence of a value in the [List<T>](#) or in a portion of it.

Overloads

 [Expand table](#)

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.

Examples

The following example demonstrates all three overloads of the [LastIndexOf](#) method. A [List<T>](#) of strings is created, with one entry that appears twice, at index location 0 and index location 5. The [LastIndexOf\(T\)](#) method overload searches the entire list from the end, and finds the second occurrence of the string. The [LastIndexOf\(T, Int32\)](#) method overload is used to search the list backward beginning with index location 3 and continuing to the beginning of the list, so it finds the first occurrence of the string in the list. Finally, the [LastIndexOf\(T, Int32, Int32\)](#) method overload is used to search a range of four entries, beginning at index location 4 and extending backward (that is, it searches the items at locations 4, 3, 2, and 1); this search returns -1 because there are no instances of the search string in that range.

C#

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

```

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        dinosaurs.Add("Tyrannosaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Brachiosaurus");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Tyrannosaurus");
        dinosaurs.Add("Compsognathus");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nLastIndexOf(\"Tyrannosaurus\"): {0}",
            dinosaurs.LastIndexOf("Tyrannosaurus"));

        Console.WriteLine("\nLastIndexOf(\"Tyrannosaurus\", 3): {0}",
            dinosaurs.LastIndexOf("Tyrannosaurus", 3));

        Console.WriteLine("\nLastIndexOf(\"Tyrannosaurus\", 4, 4): {0}",
            dinosaurs.LastIndexOf("Tyrannosaurus", 4, 4));
    }
}

/* This code example produces the following output:

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Deinonychus
Tyrannosaurus
Compsognathus

LastIndexOf("Tyrannosaurus"): 5

LastIndexOf("Tyrannosaurus", 3): 0

LastIndexOf("Tyrannosaurus", 4, 4): -1
*/

```

LastIndexOf(T)

Searches for the specified object and returns the zero-based index of the last occurrence within the entire [List<T>](#).

C#

```
public int LastIndexOf(T item);
```

Parameters

item [T](#)

The object to locate in the [List<T>](#). The value can be `null` for reference types.

Returns

[Int32](#)

The zero-based index of the last occurrence of `item` within the entire the [List<T>](#), if found; otherwise, -1.

Remarks

The [List<T>](#) is searched backward starting at the last element and ending at the first element.

This method determines equality using the default equality comparer [EqualityComparer<T>.Default](#) for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

See also

- [IndexOf\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int LastIndexOf(T item, int index);
```

Parameters

item [T](#)

The object to locate in the [List<T>](#). The value can be [null](#) for reference types.

index [Int32](#)

The zero-based starting index of the backward search.

Returns

[Int32](#)

The zero-based index of the last occurrence of [item](#) within the range of elements in the [List<T>](#) that extends from the first element to [index](#), if found; otherwise, -1.

Exceptions

[ArgumentOutOfRangeException](#)

[index](#) is outside the range of valid indexes for the [List<T>](#).

Remarks

The `List<T>` is searched backward starting at `index` and ending at the first element.

This method determines equality using the default equality comparer `EqualityComparer<T>.Default` for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is the number of elements from the beginning of the `List<T>` to `index`.

See also

- [IndexOf\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public int LastIndexOf(T item, int index, int count);
```

Parameters

`item` `T`

The object to locate in the `List<T>`. The value can be `null` for reference types.

index [Int32](#)

The zero-based starting index of the backward search.

count [Int32](#)

The number of elements in the section to search.

Returns

[Int32](#)

The zero-based index of the last occurrence of `item` within the range of elements in the `List<T>` that contains `count` number of elements and ends at `index`, if found; otherwise, -1.

Exceptions

[ArgumentOutOfRangeException](#)

`index` is outside the range of valid indexes for the `List<T>`.

-or-

`count` is less than 0.

-or-

`index` and `count` do not specify a valid section in the `List<T>`.

Remarks

The `List<T>` is searched backward starting at `index` and ending at `index` minus `count` plus 1, if `count` is greater than 0.

This method determines equality using the default equality comparer `EqualityComparer<T>.Default` for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `count`.

See also

- [IndexOf\(T\)](#)
- [Contains\(T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.Remove(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a specific object from the [List<T>](#).

C#

```
public bool Remove(T item);
```

Parameters

item T

The object to remove from the [List<T>](#). The value can be `null` for reference types.

Returns

Boolean

`true` if `item` is successfully removed; otherwise, `false`. This method also returns `false` if `item` was not found in the [List<T>](#).

Implements

[Remove\(T\)](#)

Examples

The following example demonstrates how to add, remove, and insert a simple business object in a [List<T>](#).

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify the type of part
// but the part name can change.
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }
```

```

public int PartId { get; set; }

public override string ToString()
{
    return "ID: " + PartId + "    Name: " + PartName;
}
public override bool Equals(object obj)
{
    if (obj == null) return false;
    Part objAsPart = obj as Part;
    if (objAsPart == null) return false;
    else return Equals(objAsPart);
}
public override int GetHashCode()
{
    return PartId;
}
public bool Equals(Part other)
{
    if (other == null) return false;
    return (this.PartId.Equals(other.PartId));
}
// Should also override == and != operators.
}

public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        // Write out the parts in the list. This will call the overridden ToString
method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Check the list for part #1734. This calls the IEquatable.Equals method
        // of the Part class, which checks the PartId for equality.
        Console.WriteLine("\nContains(\"1734\"): {0}",
parts.Contains(new Part { PartId = 1734, PartName = "" }));

        // Insert a new item at position 2.
    }
}

```

```
Console.WriteLine("\nInsert(2, \"1834\"));  
parts.Insert(2, new Part() { PartName = "brake lever", PartId = 1834 });  
  
//Console.WriteLine();  
foreach (Part aPart in parts)  
{  
    Console.WriteLine(aPart);  
}  
  
Console.WriteLine("\nParts[3]: {0}", parts[3]);  
  
Console.WriteLine("\nRemove(\"1534\"));  
  
// This will remove part 1534 even though the PartName is different,  
// because the Equals method only checks PartId for equality.  
parts.Remove(new Part() { PartId = 1534, PartName = "cogs" });  
  
Console.WriteLine();  
foreach (Part aPart in parts)  
{  
    Console.WriteLine(aPart);  
}  
Console.WriteLine("\nRemoveAt(3)");  
// This will remove the part at index 3.  
parts.RemoveAt(3);  
  
Console.WriteLine();  
foreach (Part aPart in parts)  
{  
    Console.WriteLine(aPart);  
}  
  
/*  
  
ID: 1234  Name: crank arm  
ID: 1334  Name: chain ring  
ID: 1434  Name: regular seat  
ID: 1444  Name: banana seat  
ID: 1534  Name: cassette  
ID: 1634  Name: shift lever  
  
Contains("1734"): False  
  
Insert(2, "1834")  
ID: 1234  Name: crank arm  
ID: 1334  Name: chain ring  
ID: 1834  Name: brake lever  
ID: 1434  Name: regular seat  
ID: 1444  Name: banana seat  
ID: 1534  Name: cassette  
ID: 1634  Name: shift lever  
  
Parts[3]: ID: 1434  Name: regular seat  
  
Remove("1534")
```

```

        ID: 1234  Name: crank arm
        ID: 1334  Name: chain ring
        ID: 1834  Name: brake lever
        ID: 1434  Name: regular seat
        ID: 1444  Name: banana seat
        ID: 1634  Name: shift lever

    RemoveAt(3)

        ID: 1234  Name: crank arm
        ID: 1334  Name: chain ring
        ID: 1834  Name: brake lever
        ID: 1444  Name: banana seat
        ID: 1634  Name: shift lever

    */
}
}

```

The following example demonstrates [Remove](#) method. Several properties and methods of the `List<T>` generic class are used to add, insert, and search the list. After these operations, the list contains a duplicate. The [Remove](#) method is used to remove the first instance of the duplicate item, and the contents are displayed. The [Remove](#) method always removes the first instance it encounters.

C#

```

List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
dinosaurs.Insert(2, "Compsognathus");

```

```
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

// Shows accessing the list using the Item property.
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosaurs.Remove("Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

/* This code example produces the following output:
```

```
Capacity: 0
```

```
Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus
```

```
Capacity: 8
Count: 5
```

```
Contains("Deinonychus"): True
```

```
Insert(2, "Compsognathus")
```

```
Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus
```

```
dinosaurs[3]: Mamenchisaurus
```

```
Remove("Compsognathus")
```

```
Tyrannosaurus  
Amargasaurus  
Mamenchisaurus  
Deinonychus  
Compsognathus
```

```
TrimExcess()  
Capacity: 5  
Count: 5
```

```
Clear()  
Capacity: 5  
Count: 0  
*/
```

Remarks

If type `T` implements the `IEquatable<T>` generic interface, the equality comparer is the `Equals` method of that interface; otherwise, the default equality comparer is `Object.Equals`.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [RemoveAt\(Int32\)](#)
- [RemoveRange\(Int32, Int32\)](#)
- [Add\(T\)](#)
- [Insert\(Int32, T\)](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

List<T>.RemoveAll(Predicate<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all the elements that match the conditions defined by the specified predicate.

C#

```
public int RemoveAll(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions of the elements to remove.

Returns

[Int32](#)

The number of elements removed from the [List<T>](#).

Exceptions

[ArgumentNullException](#)

match is **null**.

Examples

The following example demonstrates the [RemoveAll](#) method and several other methods that use the [Predicate<T>](#) generic delegate.

A [List<T>](#) of strings is created, containing 8 dinosaur names, two of which (at positions 1 and 5) end with "saurus". The example also defines a search predicate method named [EndsWithSaurus](#), which accepts a string parameter and returns a Boolean value indicating whether the input string ends in "saurus".

The [Find](#), [FindLast](#), and [FindAll](#) methods are used to search the list with the search predicate method.

The `RemoveAll` method is used to remove all entries ending with "saurus". It traverses the list from the beginning, passing each element in turn to the `EndsWithSaurus` method. The element is removed if the `EndsWithSaurus` method returns `true`.

! Note

In C# and Visual Basic, it is not necessary to create the `Predicate<string>` delegate (`Predicate(Of String)` in Visual Basic) explicitly. These languages infer the correct delegate from context, and create it automatically.

Finally, the `Exists` method verifies that there are no strings in the list that end with "saurus".

C#

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

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        dinosaurs.Add("Compsognathus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Oviraptor");
        dinosaurs.Add("Velociraptor");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Dilophosaurus");
        dinosaurs.Add("Gallimimus");
        dinosaurs.Add("Triceratops");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nTrueForAll(EndsWithSaurus): {0}",
            dinosaurs.TrueForAll(EndsWithSaurus));

        Console.WriteLine("\nFind(EndsWithSaurus): {0}",
            dinosaurs.Find(EndsWithSaurus));

        Console.WriteLine("\nFindLast(EndsWithSaurus): {0}",
            dinosaurs.FindLast(EndsWithSaurus));

        Console.WriteLine("\nFindAll(EndsWithSaurus):");
        List<string> sublist = dinosaurs.FindAll(EndsWithSaurus);
```

```
foreach(string dinosaur in sublist)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine(
    "\n{0} elements removed by RemoveAll(EndsWithSaurus).",
    dinosaurs.RemoveAll(EndsWithSaurus));

Console.WriteLine("\nList now contains:");
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nExists(EndsWithSaurus): {0}",
    dinosaurs.Exists(EndsWithSaurus));
}

// Search predicate returns true if a string ends in "saurus".
private static bool EndsWithSaurus(String s)
{
    return s.ToLower().EndsWith("saurus");
}
```

/ This code example produces the following output:*

```
Compsognathus
Amargasaurus
Oviraptor
Velociraptor
Deinonychus
Dilophosaurus
Gallimimus
Triceratops
```

```
TrueForAll(EndsWithSaurus): False
```

```
Find(EndsWithSaurus): Amargasaurus
```

```
FindLast(EndsWithSaurus): Dilophosaurus
```

```
FindAll(EndsWithSaurus):
Amargasaurus
Dilophosaurus
```

```
2 elements removed by RemoveAll(EndsWithSaurus).
```

```
List now contains:
Compsognathus
Oviraptor
Velociraptor
Deinonychus
Gallimimus
```

Triceratops

```
Exists(EndsWithSaurus): False  
*/
```

Remarks

The [Predicate<T>](#) is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current [List<T>](#) are individually passed to the [Predicate<T>](#) delegate, and the elements that match the conditions are removed from the [List<T>](#).

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [FindAll\(Predicate<T>\)](#)
- [Predicate<T>](#)
- [Remove\(T\)](#)
- [RemoveAt\(Int32\)](#)
- [RemoveRange\(Int32, Int32\)](#)

List<T>.RemoveAt(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the element at the specified index of the [List<T>](#).

C#

```
public void RemoveAt(int index);
```

Parameters

index [Int32](#)

The zero-based index of the element to remove.

Implements

[RemoveAt\(Int32\)](#) , [RemoveAt\(Int32\)](#)

Exceptions

[ArgumentOutOfRangeException](#)

index is less than 0.

-or-

index is equal to or greater than [Count](#).

Examples

The following example demonstrates how to add, remove, and insert a simple business object in a [List<T>](#).

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify the type of part
// but the part name can change.
public class Part : IEquatable<Part>
```

```

{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public override int GetHashCode()
    {
        return PartId;
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        // Write out the parts in the list. This will call the overridden ToString
        method
        // in the Part class.
        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Check the list for part #1734. This calls the IEquatable.Equals method
        // of the Part class, which checks the PartId for equality.
        Console.WriteLine("\nContains(\"1734\"): {0}",
        parts.Contains(new Part { PartId = 1734, PartName = "" }));
    }
}

```

```

// Insert a new item at position 2.
Console.WriteLine("\nInsert(2, \"1834\")");
parts.Insert(2, new Part() { PartName = "brake lever", PartId = 1834 });

//Console.WriteLine();
foreach (Part aPart in parts)
{
    Console.WriteLine(aPart);
}

Console.WriteLine("\nParts[3]: {0}", parts[3]);

Console.WriteLine("\nRemove(\"1534\")");

// This will remove part 1534 even though the PartName is different,
// because the Equals method only checks PartId for equality.
parts.Remove(new Part() { PartId = 1534, PartName = "cogs" });

Console.WriteLine();
foreach (Part aPart in parts)
{
    Console.WriteLine(aPart);
}
Console.WriteLine("\nRemoveAt(3)");
// This will remove the part at index 3.
parts.RemoveAt(3);

Console.WriteLine();
foreach (Part aPart in parts)
{
    Console.WriteLine(aPart);
}

/*
ID: 1234  Name: crank arm
ID: 1334  Name: chain ring
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

Contains("1734"): False

Insert(2, "1834")
ID: 1234  Name: crank arm
ID: 1334  Name: chain ring
ID: 1834  Name: brake lever
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

Parts[3]: ID: 1434  Name: regular seat

```

```

        Remove("1534")

        ID: 1234  Name: crank arm
        ID: 1334  Name: chain ring
        ID: 1834  Name: brake lever
        ID: 1434  Name: regular seat
        ID: 1444  Name: banana seat
        ID: 1634  Name: shift lever

        RemoveAt(3)

        ID: 1234  Name: crank arm
        ID: 1334  Name: chain ring
        ID: 1834  Name: brake lever
        ID: 1444  Name: banana seat
        ID: 1634  Name: shift lever

    */
}

```

Remarks

When you call [RemoveAt](#) to remove an item, the remaining items in the list are renumbered to replace the removed item. For example, if you remove the item at index 3, the item at index 4 is moved to the 3 position. In addition, the number of items in the list (as represented by the [Count](#) property) is reduced by 1.

This method is an O(*n*) operation, where *n* is ([Count](#) - `index`).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- Remove(T)
- RemoveRange(Int32, Int32)
- Add(T)
- Insert(Int32, T)

List<T>.RemoveRange(Int32, Int32)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes a range of elements from the [List<T>](#).

C#

```
public void RemoveRange(int index, int count);
```

Parameters

index [Int32](#)

The zero-based starting index of the range of elements to remove.

count [Int32](#)

The number of elements to remove.

Exceptions

[ArgumentOutOfRangeException](#)

index is less than 0.

-or-

count is less than 0.

[ArgumentException](#)

index and **count** do not denote a valid range of elements in the [List<T>](#).

Examples

The following example demonstrates the [RemoveRange](#) method and various other methods of the [List<T>](#) class that act on ranges. After the list has been created and modified, the [RemoveRange](#) method is used to remove two elements from the list, beginning at index location 2.

C#

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

public class Example
{
    public static void Main()
    {
        string[] input = { "Brachiosaurus",
                           "Amargasaurus",
                           "Mamenchisaurus" };

        List<string> dinosaurs = new List<string>(input);

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nAddRange(dinosaurs)");
        dinosaurs.AddRange(dinosaurs);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nRemoveRange(2, 2)");
        dinosaurs.RemoveRange(2, 2);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }

        input = new string[] { "Tyrannosaurus",
                              "Deinonychus",
                              "Velociraptor" };

        Console.WriteLine("\nInsertRange(3, input)");
        dinosaurs.InsertRange(3, input);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
            Console.WriteLine(dinosaur);
        }
    }
}
```

```
Console.WriteLine("\noutput = dinosaurs.GetRange(2, 3).ToArray()");  
string[] output = dinosaurs.GetRange(2, 3).ToArray();  
  
Console.WriteLine();  
foreach( string dinosaur in output )  
{  
    Console.WriteLine(dinosaur);  
}  
}  
}  
  
/* This code example produces the following output:  
  
Capacity: 3  
  
Brachiosaurus  
Amargasaurus  
Mamenchisaurus  
  
AddRange(dinosaurs)  
  
Brachiosaurus  
Amargasaurus  
Mamenchisaurus  
Brachiosaurus  
Amargasaurus  
Mamenchisaurus  
  
RemoveRange(2, 2)  
  
Brachiosaurus  
Amargasaurus  
Amargasaurus  
Mamenchisaurus  
  
InsertRange(3, input)  
  
Brachiosaurus  
Amargasaurus  
Amargasaurus  
Tyrannosaurus  
Deinonychus  
Velociraptor  
Mamenchisaurus  
  
output = dinosaurs.GetRange(2, 3).ToArray()  
  
Amargasaurus  
Tyrannosaurus  
Deinonychus  
*/
```

Remarks

The items are removed and all the elements following them in the [List<T>](#) have their indexes reduced by `count`.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove\(T\)](#)
- [RemoveAt\(Int32\)](#)
- [GetRange\(Int32, Int32\)](#)
- [AddRange\(IEnumerable<T>\)](#)
- [InsertRange\(Int32, IEnumerable<T>\)](#)

List<T>.Reverse Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Reverses the order of the elements in the [List<T>](#) or a portion of it.

Overloads

[] [Expand table](#)

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.

Examples

The following example demonstrates both overloads of the [Reverse](#) method. The example creates a [List<T>](#) of strings and adds six strings. The [Reverse\(\)](#) method overload is used to reverse the list, and then the [Reverse\(Int32, Int32\)](#) method overload is used to reverse the middle of the list, beginning with element 1 and encompassing four elements.

C#

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

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        dinosaurs.Add("Pachycephalosaurus");
        dinosaurs.Add("Parasaurolophus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Coelophysis");
        dinosaurs.Add("Oviraptor");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
```

```
        Console.WriteLine(dinosaur);
    }

    dinosaurs.Reverse();

    Console.WriteLine();
    foreach(string dinosaur in dinosaurs)
    {
        Console.WriteLine(dinosaur);
    }

    dinosaurs.Reverse(1, 4);

    Console.WriteLine();
    foreach(string dinosaur in dinosaurs)
    {
        Console.WriteLine(dinosaur);
    }
}

/* This code example produces the following output:

Pachycephalosaurus
Parasaurolophus
Mamenchisaurus
Amargasaurus
Coelophysis
Oviraptor

Oviraptor
Coelophysis
Amargasaurus
Mamenchisaurus
Parasaurolophus
Pachycephalosaurus

Oviraptor
Parasaurolophus
Mamenchisaurus
Amargasaurus
Coelophysis
Pachycephalosaurus
*/
```

Reverse()

Reverses the order of the elements in the entire [List<T>](#).

C#

```
public void Reverse();
```

Remarks

This method uses [Array.Reverse](#) to reverse the order of the elements.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Reverse(Int32, Int32)

Reverses the order of the elements in the specified range.

C#

```
public void Reverse(int index, int count);
```

Parameters

index [Int32](#)

The zero-based starting index of the range to reverse.

count [Int32](#)

The number of elements in the range to reverse.

Exceptions

[ArgumentOutOfRangeException](#)

`index` is less than 0.

-or-

`count` is less than 0.

ArgumentException

`index` and `count` do not denote a valid range of elements in the [List<T>](#).

Remarks

This method uses [Array.Reverse](#) to reverse the order of the elements.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.Sort Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sorts the elements or a portion of the elements in the [List<T>](#) using either the specified or default [IComparer<T>](#) implementation or a provided [Comparison<T>](#) delegate to compare list elements.

Overloads

 [Expand table](#)

Sort(Comparison<T>)	Sorts the elements in the entire List<T> using the specified Comparison<T> .
Sort(Int32, Int32, IComparer<T>)	Sorts the elements in a range of elements in List<T> using the specified comparer.
Sort()	Sorts the elements in the entire List<T> using the default comparer.
Sort(IComparer<T>)	Sorts the elements in the entire List<T> using the specified comparer.

Sort(Comparison<T>)

Sorts the elements in the entire [List<T>](#) using the specified [Comparison<T>](#).

C#

```
public void Sort(Comparison<T> comparison);
```

Parameters

comparison [Comparison<T>](#)

The [Comparison<T>](#) to use when comparing elements.

Exceptions

[ArgumentNullException](#)

`comparison` is `null`.

ArgumentException

The implementation of `comparison` caused an error during the sort. For example, `comparison` might not return 0 when comparing an item with itself.

Examples

The following code demonstrates the `Sort` and `Sort` method overloads on a simple business object. Calling the `Sort` method results in the use of the default comparer for the `Part` type, and the `Sort` method is implemented using an anonymous method.

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify the type of part
// but the part name can change.
public class Part : IEquatable<Part>, IComparable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public int SortByNameAscending(string name1, string name2)
    {

        return name1.CompareTo(name2);
    }

    // Default comparer for Part type.
    public int CompareTo(Part comparePart)
    {
        // A null value means that this object is greater.
        if (comparePart == null)
            return 1;

        else
            return this.PartId.CompareTo(comparePart.PartId);
    }
}
```

```
    }
    public override int GetHashCode()
    {
        return PartId;
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName= "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 } );
        // Name intentionally left null.
        parts.Add(new Part() { PartId = 1334 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });

        // Write out the parts in the list. This will call the overridden
        // ToString method in the Part class.
        Console.WriteLine("\nBefore sort:");
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Call Sort on the list. This will use the
        // default comparer, which is the Compare method
        // implemented on Part.
        parts.Sort();

        Console.WriteLine("\nAfter sort by part number:");
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // This shows calling the Sort(Comparison(T) overload using
        // an anonymous method for the Comparison delegate.
        // This method treats null as the lesser of two values.
        parts.Sort(delegate(Part x, Part y)
        {
            if (x.PartName == null && y.PartName == null) return 0;
            else if (x.PartName == null) return -1;
            else if (y.PartName == null) return 1;
        });
    }
}
```

```

        else return x.PartName.CompareTo(y.PartName);
    });

    Console.WriteLine("\nAfter sort by name:");
    foreach (Part aPart in parts)
    {
        Console.WriteLine(aPart);
    }

/*
    Before sort:
ID: 1434  Name: regular seat
ID: 1234  Name: crank arm
ID: 1634  Name: shift lever
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette

    After sort by part number:
ID: 1234  Name: crank arm
ID: 1334  Name:
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

    After sort by name:
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1234  Name: crank arm
ID: 1434  Name: regular seat
ID: 1634  Name: shift lever

*/
}
}

```

The following example demonstrates the [Sort\(Comparison<T>\)](#) method overload.

The example defines an alternative comparison method for strings, named [CompareDinosByLength](#). This method works as follows: First, the comparands are tested for `null`, and a null reference is treated as less than a non-null. Second, the string lengths are compared, and the longer string is deemed to be greater. Third, if the lengths are equal, ordinary string comparison is used.

A [List<T>](#) of strings is created and populated with four strings, in no particular order. The list also includes an empty string and a null reference. The list is displayed, sorted using a [Comparison<T>](#) generic delegate representing the [CompareDinosByLength](#) method, and displayed again.

C#

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

public class Example
{
    private static int CompareDinosByLength(string x, string y)
    {
        if (x == null)
        {
            if (y == null)
            {
                // If x is null and y is null, they're
                // equal.
                return 0;
            }
            else
            {
                // If x is null and y is not null, y
                // is greater.
                return -1;
            }
        }
        else
        {
            // If x is not null...
            //
            if (y == null)
                // ...and y is null, x is greater.
            {
                return 1;
            }
            else
            {
                // ...and y is not null, compare the
                // lengths of the two strings.
                //
                int retval = x.Length.CompareTo(y.Length);

                if (retval != 0)
                {
                    // If the strings are not of equal length,
                    // the longer string is greater.
                    //
                    return retval;
                }
                else
                {
                    // If the strings are of equal length,
                    // sort them with ordinary string comparison.
                    //
                    return x.CompareTo(y);
                }
            }
        }
    }
}
```

```

        }
    }

public static void Main()
{
    List<string> dinosaurs = new List<string>();
    dinosaurs.Add("Pachycephalosaurus");
    dinosaurs.Add("Amargasaurus");
    dinosaurs.Add("");
    dinosaurs.Add(null);
    dinosaurs.Add("Mamenchisaurus");
    dinosaurs.Add("Deinonychus");
    Display(dinosaurs);

    Console.WriteLine("\nSort with generic Comparison<string> delegate:");
    dinosaurs.Sort(CompareDinosByLength);
    Display(dinosaurs);
}

private static void Display(List<string> list)
{
    Console.WriteLine();
    foreach( string s in list )
    {
        if (s == null)
            Console.WriteLine("(null)");
        else
            Console.WriteLine("\'{0}\'", s);
    }
}
}

/* This code example produces the following output:

"Pachycephalosaurus"
"Amargasaurus"
""
(null)
"Mamenchisaurus"
"Deinonychus"

Sort with generic Comparison<string> delegate:

(null)
""
"Deinonychus"
"Amargasaurus"
"Mamenchisaurus"
"Pachycephalosaurus"
*/

```

Remarks

If `comparison` is provided, the elements of the `List<T>` are sorted using the method represented by the delegate.

If `comparison` is `null`, an [ArgumentNullException](#) is thrown.

This method uses [Array.Sort](#), which applies the introspective sort as follows:

- If the partition size is less than or equal to 16 elements, it uses an insertion sort algorithm
- If the number of partitions exceeds $2 \log n$, where n is the range of the input array, it uses a [Heapsort](#) algorithm.
- Otherwise, it uses a Quicksort algorithm.

This implementation performs an unstable sort; that is, if two elements are equal, their order might not be preserved. In contrast, a stable sort preserves the order of elements that are equal.

This method is an $O(n \log n)$ operation, where n is [Count](#).

See also

- [Comparison<T>](#)
- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sort(Int32, Int32, IComparer<T>)

Sorts the elements in a range of elements in `List<T>` using the specified comparer.

C#

```
public void Sort(int index, int count, System.Collections.Generic.IComparer<T>? comparer);
```

Parameters

index `Int32`

The zero-based starting index of the range to sort.

count `Int32`

The length of the range to sort.

comparer `IComparer<T>`

The `IComparer<T>` implementation to use when comparing elements, or `null` to use the default comparer `Default`.

Exceptions

[ArgumentOutOfRangeException](#)

`index` is less than 0.

-or-

`count` is less than 0.

[ArgumentException](#)

`index` and `count` do not specify a valid range in the `List<T>`.

-or-

The implementation of `comparer` caused an error during the sort. For example, `comparer` might not return 0 when comparing an item with itself.

[InvalidOperationException](#)

`comparer` is `null`, and the default comparer `Default` cannot find implementation of the `IComparable<T>` generic interface or the `IComparable` interface for type `T`.

Examples

The following example demonstrates the `Sort(Int32, Int32, IComparer<T>)` method overload and the `BinarySearch(Int32, Int32, T, IComparer<T>)` method overload.

The example defines an alternative comparer for strings named DinoComparer, which implements the `IComparer<string>` (`IComparer(Of String)` in Visual Basic) generic interface. The comparer works as follows: First, the comparands are tested for `null`, and a null reference is treated as less than a non-null. Second, the string lengths are compared, and the longer string is deemed to be greater. Third, if the lengths are equal, ordinary string comparison is used.

A `List<T>` of strings is created and populated with the names of five herbivorous dinosaurs and three carnivorous dinosaurs. Within each of the two groups, the names are not in any particular sort order. The list is displayed, the range of herbivores is sorted using the alternate comparer, and the list is displayed again.

The `BinarySearch(Int32, Int32, T, IComparer<T>)` method overload is then used to search only the range of herbivores for "Brachiosaurus". The string is not found, and the bitwise complement (the `~` operator in C#, `xor -1` in Visual Basic) of the negative number returned by the `BinarySearch(Int32, Int32, T, IComparer<T>)` method is used as an index for inserting the new string.

C#

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

public class DinoComparer: IComparer<string>
{
    public int Compare(string x, string y)
    {
        if (x == null)
        {
            if (y == null)
            {
                // If x is null and y is null, they're
                // equal.
                return 0;
            }
            else
            {
                // If x is null and y is not null, y
                // is greater.
                return -1;
            }
        }
        else
        {
            // If x is not null...
            //
            if (y == null)
                // ...and y is null, x is greater.
            {

```

```

        return 1;
    }
    else
    {
        // ...and y is not null, compare the
        // lengths of the two strings.
        //
        int retval = x.Length.CompareTo(y.Length);

        if (retval != 0)
        {
            // If the strings are not of equal length,
            // the longer string is greater.
            //
            return retval;
        }
        else
        {
            // If the strings are of equal length,
            // sort them with ordinary string comparison.
            //
            return x.CompareTo(y);
        }
    }
}

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        dinosaurs.Add("Pachycephalosaurus");
        dinosaurs.Add("Parasauralophus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Galimimus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Oviraptor");
        dinosaurs.Add("Tyrannosaurus");

        int herbivores = 5;
        Display(dinosaurs);

        DinoComparer dc = new DinoComparer();

        Console.WriteLine("\nSort a range with the alternate comparer:");
        dinosaurs.Sort(0, herbivores, dc);
        Display(dinosaurs);

        Console.WriteLine("\nBinarySearch a range and Insert \"{0}\":",
            "Brachiosaurus");
    }
}

```

```
        int index = dinosaurs.BinarySearch(0, herbivores, "Brachiosaurus", dc);

        if (index < 0)
        {
            dinosaurs.Insert(~index, "Brachiosaurus");
            herbivores++;
        }

        Display(dinosaurs);
    }

private static void Display(List<string> list)
{
    Console.WriteLine();
    foreach( string s in list )
    {
        Console.WriteLine(s);
    }
}
}
```

/ This code example produces the following output:*

Pachycephalosaurus
Parasauralophus
Amargasaurus
Galimimus
Mamenchisaurus
Deinonychus
Oviraptor
Tyrannosaurus

Sort a range with the alternate comparer:

Galimimus
Amargasaurus
Mamenchisaurus
Parasauralophus
Pachycephalosaurus
Deinonychus
Oviraptor
Tyrannosaurus

BinarySearch a range and Insert "Brachiosaurus":

Galimimus
Amargasaurus
Brachiosaurus
Mamenchisaurus
Parasauralophus
Pachycephalosaurus
Deinonychus
Oviraptor
Tyrannosaurus
*/

Remarks

If `comparer` is provided, the elements of the `List<T>` are sorted using the specified `IComparer<T>` implementation.

If `comparer` is `null`, the default comparer `Comparer<T>.Default` checks whether type `T` implements the `IComparable<T>` generic interface and uses that implementation, if available. If not, `Comparer<T>.Default` checks whether type `T` implements the `IComparable` interface. If type `T` does not implement either interface, `Comparer<T>.Default` throws an `InvalidOperationException`.

This method uses `Array.Sort`, which applies the introspective sort as follows:

- If the partition size is less than or equal to 16 elements, it uses an insertion sort algorithm
- If the number of partitions exceeds $2 \log n$, where n is the range of the input array, it uses a [Heapsort](#) algorithm.
- Otherwise, it uses a Quicksort algorithm.

This implementation performs an unstable sort; that is, if two elements are equal, their order might not be preserved. In contrast, a stable sort preserves the order of elements that are equal.

This method is an $O(n \log n)$ operation, where n is `Count`.

See also

- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

Sort()

Sorts the elements in the entire `List<T>` using the default comparer.

C#

```
public void Sort();
```

Exceptions

InvalidOperationException

The default comparer `Default` cannot find an implementation of the `IComparable<T>` generic interface or the `IComparable` interface for type `T`.

Examples

The following example adds some names to a `List<String>` object, displays the list in unsorted order, calls the `Sort` method, and then displays the sorted list.

C#

```
String[] names = { "Samuel", "Dakota", "Koani", "Saya", "Vanya", "Jody",
                   "Yiska", "Yuma", "Jody", "Nikita" };
var nameList = new List<String>();
nameList.AddRange(names);
Console.WriteLine("List in unsorted order: ");
foreach (var name in nameList)
    Console.Write(" {0}", name);

Console.WriteLine(Environment.NewLine);

nameList.Sort();
Console.WriteLine("List in sorted order: ");
foreach (var name in nameList)
    Console.Write(" {0}", name);

Console.WriteLine();

// The example displays the following output:
//      List in unsorted order:
//          Samuel Dakota Koani Saya Vanya Jody Yiska Yuma Jody
//          Nikita
```

```
//  
//      List in sorted order:  
//          Dakota    Jody    Jody    Koani    Nikita    Samuel    Saya    Vanya    Yiska  
Yuma
```

The following code demonstrates the `Sort()` and `Sort(Comparison<T>)` method overloads on a simple business object. Calling the `Sort()` method results in the use of the default comparer for the Part type, and the `Sort(Comparison<T>)` method is implemented by using an anonymous method.

C#

```
using System;  
using System.Collections.Generic;  
// Simple business object. A PartId is used to identify the type of part  
// but the part name can change.  
public class Part : IEquatable<Part> , IComparable<Part>  
{  
    public string PartName { get; set; }  
  
    public int PartId { get; set; }  
  
    public override string ToString()  
    {  
        return "ID: " + PartId + "    Name: " + PartName;  
    }  
    public override bool Equals(object obj)  
    {  
        if (obj == null) return false;  
        Part objAsPart = obj as Part;  
        if (objAsPart == null) return false;  
        else return Equals(objAsPart);  
    }  
    public int SortByNameAscending(string name1, string name2)  
    {  
  
        return name1.CompareTo(name2);  
    }  
  
    // Default comparer for Part type.  
    public int CompareTo(Part comparePart)  
    {  
        // A null value means that this object is greater.  
        if (comparePart == null)  
            return 1;  
  
        else  
            return this.PartId.CompareTo(comparePart.PartId);  
    }  
    public override int GetHashCode()  
    {  
        return PartId;
```

```
    }
    public bool Equals(Part other)
    {
        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        List<Part> parts = new List<Part>();

        // Add parts to the list.
        parts.Add(new Part() { PartName = "regular seat", PartId = 1434 });
        parts.Add(new Part() { PartName= "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 } );
        // Name intentionally left null.
        parts.Add(new Part() { PartId = 1334 });
        parts.Add(new Part() { PartName = "banana seat", PartId = 1444 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });

        // Write out the parts in the list. This will call the overridden
        // ToString method in the Part class.
        Console.WriteLine("\nBefore sort:");
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // Call Sort on the list. This will use the
        // default comparer, which is the Compare method
        // implemented on Part.
        parts.Sort();

        Console.WriteLine("\nAfter sort by part number:");
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        // This shows calling the Sort(Comparison(T) overload using
        // an anonymous method for the Comparison delegate.
        // This method treats null as the lesser of two values.
        parts.Sort(delegate(Part x, Part y)
        {
            if (x.PartName == null && y.PartName == null) return 0;
            else if (x.PartName == null) return -1;
            else if (y.PartName == null) return 1;
            else return x.PartName.CompareTo(y.PartName);
        });
        Console.WriteLine("\nAfter sort by name:");
    }
}
```

```

        foreach (Part aPart in parts)
    {
        Console.WriteLine(aPart);
    }

/*
    Before sort:
ID: 1434  Name: regular seat
ID: 1234  Name: crank arm
ID: 1634  Name: shift lever
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette

    After sort by part number:
ID: 1234  Name: crank arm
ID: 1334  Name:
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

    After sort by name:
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1234  Name: crank arm
ID: 1434  Name: regular seat
ID: 1634  Name: shift lever

*/
}
}

```

The following example demonstrates the [Sort\(\)](#) method overload and the [BinarySearch\(T\)](#) method overload. A [List<T>](#) of strings is created and populated with four strings, in no particular order. The list is displayed, sorted, and displayed again.

The [BinarySearch\(T\)](#) method overload is then used to search for two strings that are not in the list, and the [Insert](#) method is used to insert them. The return value of the [BinarySearch](#) method is negative in each case, because the strings are not in the list. Taking the bitwise complement (the `~` operator in C#, `xor -1` in Visual Basic) of this negative number produces the index of the first element in the list that is larger than the search string, and inserting at this location preserves the sort order. The second search string is larger than any element in the list, so the insertion position is at the end of the list.

```
List<string> dinosaurs = new List<string>();

dinosaurs.Add("Pachycephalosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");

Console.WriteLine("Initial list:");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nSort:");
dinosaurs.Sort();

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nBinarySearch and Insert \"Coelophysis\":");
int index = dinosaurs.BinarySearch("Coelophysis");
if (index < 0)
{
    dinosaurs.Insert(~index, "Coelophysis");
}

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nBinarySearch and Insert \"Tyrannosaurus\"+");
index = dinosaurs.BinarySearch("Tyrannosaurus");
if (index < 0)
{
    dinosaurs.Insert(~index, "Tyrannosaurus");
}

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}
```

/ This code example produces the following output:*

Initial list:

Pachycephalosaurus
Amargasaurus

```
Mamenchisaurus  
Deinonychus
```

Sort:

```
Amargasaurus  
Deinonychus  
Mamenchisaurus  
Pachycephalosaurus
```

BinarySearch and Insert "Coelophysis":

```
Amargasaurus  
Coelophysis  
Deinonychus  
Mamenchisaurus  
Pachycephalosaurus
```

BinarySearch and Insert "Tyrannosaurus":

```
Amargasaurus  
Coelophysis  
Deinonychus  
Mamenchisaurus  
Pachycephalosaurus  
Tyrannosaurus  
*/
```

Remarks

This method uses the default comparer [Comparer<T>.Default](#) for type `T` to determine the order of list elements. The [Comparer<T>.Default](#) property checks whether type `T` implements the [IComparable<T>](#) generic interface and uses that implementation, if available. If not, [Comparer<T>.Default](#) checks whether type `T` implements the [IComparable](#) interface. If type `T` does not implement either interface, [Comparer<T>.Default](#) throws an [InvalidOperationException](#).

This method uses the [Array.Sort](#) method, which applies the introspective sort as follows:

- If the partition size is less than or equal to 16 elements, it uses an insertion sort algorithm.
- If the number of partitions exceeds $2 \log n$, where n is the range of the input array, it uses a Heapsort algorithm.
- Otherwise, it uses a Quicksort algorithm.

This implementation performs an unstable sort; that is, if two elements are equal, their order might not be preserved. In contrast, a stable sort preserves the order of elements that are equal.

This method is an $O(n \log n)$ operation, where n is [Count](#).

See also

- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sort(IComparer<T>)

Sorts the elements in the entire [List<T>](#) using the specified comparer.

C#

```
public void Sort(System.Collections.Generic.IComparer<T>? comparer);
```

Parameters

comparer [IComparer<T>](#)

The [IComparer<T>](#) implementation to use when comparing elements, or `null` to use the default comparer [Default](#).

Exceptions

[InvalidOperationException](#)

`comparer` is `null`, and the default comparer `Default` cannot find implementation of the `IComparable<T>` generic interface or the `IComparable` interface for type `T`.

ArgumentException

The implementation of `comparer` caused an error during the sort. For example, `comparer` might not return 0 when comparing an item with itself.

Examples

The following example demonstrates the `Sort(IComparer<T>)` method overload and the `BinarySearch(T, IComparer<T>)` method overload.

The example defines an alternative comparer for strings named `DinoCompare`, which implements the `IComparer<string>` (`IComparer(Of String)` in Visual Basic) generic interface. The comparer works as follows: First, the comparands are tested for `null`, and a null reference is treated as less than a non-null. Second, the string lengths are compared, and the longer string is deemed to be greater. Third, if the lengths are equal, ordinary string comparison is used.

A `List<T>` of strings is created and populated with four strings, in no particular order. The list is displayed, sorted using the alternate comparer, and displayed again.

The `BinarySearch(T, IComparer<T>)` method overload is then used to search for several strings that are not in the list, employing the alternate comparer. The `Insert` method is used to insert the strings. These two methods are located in the function named `SearchAndInsert`, along with code to take the bitwise complement (the `~` operator in C#, `Xor -1` in Visual Basic) of the negative number returned by `BinarySearch(T, IComparer<T>)` and use it as an index for inserting the new string.

C#

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

public class DinoComparer: IComparer<string>
{
    public int Compare(string x, string y)
    {
        if (x == null)
        {
            if (y == null)
            {
                // If x is null and y is null, they're
                // equal.
                return 0;
            }
            else
                return 1;
        }
        else if (y == null)
            return -1;
        else
        {
            int lenX = x.Length;
            int lenY = y.Length;
            if (lenX < lenY)
                return -1;
            else if (lenX > lenY)
                return 1;
            else
            {
                int result = string.Compare(x, y);
                if (result != 0)
                    return result;
                else
                    return 0;
            }
        }
    }
}
```

```

        }
        else
        {
            // If x is null and y is not null, y
            // is greater.
            return -1;
        }
    }
    else
    {
        // If x is not null...
        //
        if (y == null)
            // ...and y is null, x is greater.
        {
            return 1;
        }
        else
        {
            // ...and y is not null, compare the
            // lengths of the two strings.
            //
            int retval = x.Length.CompareTo(y.Length);

            if (retval != 0)
            {
                // If the strings are not of equal length,
                // the longer string is greater.
                //
                return retval;
            }
            else
            {
                // If the strings are of equal length,
                // sort them with ordinary string comparison.
                //
                return x.CompareTo(y);
            }
        }
    }
}

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();
        dinosaurs.Add("Pachycephalosaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Deinonychus");
        Display(dinosaurs);

        DinoComparer dc = new DinoComparer();
    }
}

```

```

Console.WriteLine("\nSort with alternate comparer:");
dinosaurs.Sort(dc);
Display(dinosaurs);

SearchAndInsert(dinosaurs, "Coelophysis", dc);
Display(dinosaurs);

SearchAndInsert(dinosaurs, "Oviraptor", dc);
Display(dinosaurs);

SearchAndInsert(dinosaurs, "Tyrannosaur", dc);
Display(dinosaurs);

SearchAndInsert(dinosaurs, null, dc);
Display(dinosaurs);
}

private static void SearchAndInsert(List<string> list,
    string insert, DinoComparer dc)
{
    Console.WriteLine("\nBinarySearch and Insert \">{0}\":", insert);

    int index = list.BinarySearch(insert, dc);

    if (index < 0)
    {
        list.Insert(~index, insert);
    }
}

private static void Display(List<string> list)
{
    Console.WriteLine();
    foreach( string s in list )
    {
        Console.WriteLine(s);
    }
}
}

/* This code example produces the following output:

```

Pachycephalosaurus
Amargasaurus
Mamenchisaurus
Deinonychus

Sort with alternate comparer:

Deinonychus
Amargasaurus
Mamenchisaurus
Pachycephalosaurus

```
BinarySearch and Insert "Coelophysis":
```

```
Coelophysis
Deinonychus
Amargasaurus
Mamenchisaurus
Pachycephalosaurus
```

```
BinarySearch and Insert "Oviraptor":
```

```
Oviraptor
Coelophysis
Deinonychus
Amargasaurus
Mamenchisaurus
Pachycephalosaurus
```

```
BinarySearch and Insert "Tyrannosaur":
```

```
Oviraptor
Coelophysis
Deinonychus
Tyrannosaur
Amargasaurus
Mamenchisaurus
Pachycephalosaurus
```

```
BinarySearch and Insert "":
```

```
Oviraptor
Coelophysis
Deinonychus
Tyrannosaur
Amargasaurus
Mamenchisaurus
Pachycephalosaurus
*/
```

Remarks

If `comparer` is provided, the elements of the `List<T>` are sorted using the specified `IComparer<T>` implementation.

If `comparer` is `null`, the default comparer `Comparer<T>.Default` checks whether type `T` implements the `IComparable<T>` generic interface and uses that implementation, if available. If not, `Comparer<T>.Default` checks whether type `T` implements the `IComparable` interface. If type `T` does not implement either interface, `Comparer<T>.Default` throws an `InvalidOperationException`.

This method uses the [Array.Sort](#) method, which applies the introspective sort as follows:

- If the partition size is less than or equal to 16 elements, it uses an insertion sort algorithm.
- If the number of partitions exceeds $2 \log n$, where n is the range of the input array, it uses a Heapsort algorithm.
- Otherwise, it uses a Quicksort algorithm.

This implementation performs an unstable sort; that is, if two elements are equal, their order might not be preserved. In contrast, a stable sort preserves the order of elements that are equal.

This method is an $O(n \log n)$ operation, where n is [Count](#).

See also

- [Performing Culture-Insensitive String Operations in Collections](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.ToArray Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [List<T>](#) to a new array.

C#

```
public T[] ToArray();
```

Returns

T[]

An array containing copies of the elements of the [List<T>](#).

Examples

The following example demonstrates the [ToArray](#) method and other methods of the [List<T>](#) class that act on ranges. At the end of the example, the [GetRange](#) method is used to get three items from the list, beginning with index location 2. The [ToArray](#) method is called on the resulting [List<T>](#), creating an array of three elements. The elements of the array are displayed.

C#

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

public class Example
{
    public static void Main()
    {
        string[] input = { "Brachiosaurus",
                           "Amargasaurus",
                           "Mamenchisaurus" };

        List<string> dinosaurs = new List<string>(input);

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

        Console.WriteLine();
        foreach( string dinosaur in dinosaurs )
        {
```

```
        Console.WriteLine(dinosaur);
    }

    Console.WriteLine("\nAddRange(dinosaurs)");
    dinosaurs.AddRange(dinosaurs);

    Console.WriteLine();
    foreach( string dinosaur in dinosaurs )
    {
        Console.WriteLine(dinosaur);
    }

    Console.WriteLine("\nRemoveRange(2, 2)");
    dinosaurs.RemoveRange(2, 2);

    Console.WriteLine();
    foreach( string dinosaur in dinosaurs )
    {
        Console.WriteLine(dinosaur);
    }

    input = new string[] { "Tyrannosaurus",
                          "Deinonychus",
                          "Velociraptor" };

    Console.WriteLine("\nInsertRange(3, input)");
    dinosaurs.InsertRange(3, input);

    Console.WriteLine();
    foreach( string dinosaur in dinosaurs )
    {
        Console.WriteLine(dinosaur);
    }

    Console.WriteLine("\noutput = dinosaurs.GetRange(2, 3).ToArray()");
    string[] output = dinosaurs.GetRange(2, 3).ToArray();

    Console.WriteLine();
    foreach( string dinosaur in output )
    {
        Console.WriteLine(dinosaur);
    }
}

/* This code example produces the following output:

Capacity: 3

Brachiosaurus
Amargasaurus
Mamenchisaurus

AddRange(dinosaurs)
```

```

Brachiosaurus
Amargasaurus
Mamenchisaurus
Brachiosaurus
Amargasaurus
Mamenchisaurus

RemoveRange(2, 2)

Brachiosaurus
Amargasaurus
Amargasaurus
Mamenchisaurus

InsertRange(3, input)

Brachiosaurus
Amargasaurus
Amargasaurus
Tyrannosaurus
Deinonychus
Velociraptor
Mamenchisaurus

output = dinosaurs.GetRange(2, 3).ToArray()

Amargasaurus
Tyrannosaurus
Deinonychus
*/

```

Remarks

The elements are copied using [Array.Copy](#), which is an O(n) operation, where n is [Count](#).

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.TrimExcess Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the capacity to the actual number of elements in the [List<T>](#), if that number is less than a threshold value.

C#

```
public void TrimExcess();
```

Examples

The following example demonstrates how to check the capacity and count of a [List<T>](#) that contains a simple business object, and illustrates using the [TrimExcess](#) method to remove extra capacity.

C#

```
using System;
using System.Collections.Generic;
// Simple business object. A PartId is used to identify a part
// but the part name be different for the same Id.
public class Part : IEquatable<Part>
{
    public string PartName { get; set; }
    public int PartId { get; set; }
    public override string ToString()
    {
        return "ID: " + PartId + "    Name: " + PartName;
    }
    public override bool Equals(object obj)
    {
        if (obj == null) return false;
        Part objAsPart = obj as Part;
        if (objAsPart == null) return false;
        else return Equals(objAsPart);
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
    public bool Equals(Part other)
    {
```

```

        if (other == null) return false;
        return (this.PartId.Equals(other.PartId));
    }
    // Should also override == and != operators.
}
public class Example
{

    public static void Main()
    {
        List<Part> parts = new List<Part>();

        Console.WriteLine("\nCapacity: {0}", parts.Capacity);

        parts.Add(new Part() { PartName = "crank arm", PartId = 1234 });
        parts.Add(new Part() { PartName = "chain ring", PartId = 1334 });
        parts.Add(new Part() { PartName = "seat", PartId = 1434 });
        parts.Add(new Part() { PartName = "cassette", PartId = 1534 });
        parts.Add(new Part() { PartName = "shift lever", PartId = 1634 });

        Console.WriteLine();
        foreach (Part aPart in parts)
        {
            Console.WriteLine(aPart);
        }

        Console.WriteLine("\nCapacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);

        parts.TrimExcess();
        Console.WriteLine("\nTrimExcess()");
        Console.WriteLine("Capacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);

        parts.Clear();
        Console.WriteLine("\nClear()");
        Console.WriteLine("Capacity: {0}", parts.Capacity);
        Console.WriteLine("Count: {0}", parts.Count);
    }
    /*
     * This code example produces the following output.
     * Capacity: 0

     * ID: 1234  Name: crank arm
     * ID: 1334  Name: chain ring
     * ID: 1434  Name: seat
     * ID: 1534  Name: cassette
     * ID: 1634  Name: shift lever

     * Capacity: 8
     * Count: 5

     * TrimExcess()
     * Capacity: 5
     * Count: 5
    */
}

```

```
        Clear()
        Capacity: 5
        Count: 0
    */
}
```

The following example demonstrates the [TrimExcess](#) method. Several properties and methods of the `List<T>` class are used to add, insert, and remove items from a list of strings. Then the [TrimExcess](#) method is used to reduce the capacity to match the count, and the [Capacity](#) and [Count](#) properties are displayed. If the unused capacity had been less than 10 percent of total capacity, the list would not have been resized. Finally, the contents of the list are cleared.

C#

```
List<string> dinosaurs = new List<string>();

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");
Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
dinosaurs.Insert(2, "Compsognathus");

Console.WriteLine();
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

// Shows accessing the list using the Item property.
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosaurs.Remove("Compsognathus");

Console.WriteLine();
```

```
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);

dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
Console.WriteLine("Count: {0}", dinosaurs.Count);
```

/* This code example produces the following output:

Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8

Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosaurs[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

TrimExcess()
Capacity: 5
Count: 5

Clear()
Capacity: 5

```
Count: 0  
*/
```

Remarks

This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large [List<T>](#) can be considerable, however, so the [TrimExcess](#) method does nothing if the list is at more than 90 percent of capacity. This avoids incurring a large reallocation cost for a relatively small gain.

ⓘ Note

The current threshold of 90 percent might change in future releases.

This method is an $O(n)$ operation, where n is [Count](#).

To reset a [List<T>](#) to its initial state, call the [Clear](#) method before calling the [TrimExcess](#) method. Trimming an empty [List<T>](#) sets the capacity of the [List<T>](#) to the default capacity.

The capacity can also be set using the [Capacity](#) property.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Clear\(\)](#)
- [Capacity](#)
- [Count](#)

List<T>.TrueForAll(Predicate<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether every element in the [List<T>](#) matches the conditions defined by the specified predicate.

C#

```
public bool TrueForAll(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The [Predicate<T>](#) delegate that defines the conditions to check against the elements.

Returns

[Boolean](#)

`true` if every element in the [List<T>](#) matches the conditions defined by the specified predicate; otherwise, `false`. If the list has no elements, the return value is `true`.

Exceptions

[ArgumentNullException](#)

`match` is `null`.

Examples

The following example demonstrates the [TrueForAll](#) method and several other methods that use [Predicate<T>](#) generic delegate.

A [List<T>](#) of strings is created, containing 8 dinosaur names, two of which (at positions 1 and 5) end with "saurus". The example also defines a search predicate method named `EndsWithSaurus`, which accepts a string parameter and returns a Boolean value indicating whether the input string ends in "saurus".

The `TrueForAll` method traverses the list from the beginning, passing each element in turn to the `EndsWithSaurus` method. The search stops when the `EndsWithSaurus` method returns `false`.

! Note

In C# and Visual Basic, it is not necessary to create the `Predicate<string>` delegate (`Predicate(Of String)` in Visual Basic) explicitly. These languages infer the correct delegate from context and create it automatically.

C#

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

public class Example
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        dinosaurs.Add("Compsognathus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Oviraptor");
        dinosaurs.Add("Velociraptor");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Dilophosaurus");
        dinosaurs.Add("Gallimimus");
        dinosaurs.Add("Triceratops");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nTrueForAll(EndsWithSaurus): {0}",
            dinosaurs.TrueForAll(EndsWithSaurus));

        Console.WriteLine("\nFind(EndsWithSaurus): {0}",
            dinosaurs.Find(EndsWithSaurus));

        Console.WriteLine("\nFindLast(EndsWithSaurus): {0}",
            dinosaurs.FindLast(EndsWithSaurus));

        Console.WriteLine("\nFindAll(EndsWithSaurus):");
        List<string> sublist = dinosaurs.FindAll(EndsWithSaurus);

        foreach(string dinosaur in sublist)
        {
            Console.WriteLine(dinosaur);
        }
    }
}
```

```
Console.WriteLine(
    "\n{0} elements removed by RemoveAll(EndsWithSaurus).",
    dinosaurs.RemoveAll(EndsWithSaurus));

Console.WriteLine("\nList now contains:");
foreach(string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

Console.WriteLine("\nExists(EndsWithSaurus): {0}",
    dinosaurs.Exists(EndsWithSaurus));
}

// Search predicate returns true if a string ends in "saurus".
private static bool EndsWithSaurus(String s)
{
    return s.ToLower().EndsWith("saurus");
}
```

/* This code example produces the following output:

```
Compsognathus
Amargasaurus
Oviraptor
Velociraptor
Deinonychus
Dilophosaurus
Gallimimus
Triceratops
```

TrueForAll(EndsWithSaurus): False

Find(EndsWithSaurus): Amargasaurus

FindLast(EndsWithSaurus): Dilophosaurus

FindAll(EndsWithSaurus):

```
Amargasaurus
Dilophosaurus
```

2 elements removed by RemoveAll(EndsWithSaurus).

List now contains:

```
Compsognathus
Oviraptor
Velociraptor
Deinonychus
Gallimimus
Triceratops
```

Exists(EndsWithSaurus): False

*/

Remarks

The [Predicate<T>](#) is a delegate to a method that returns `true` if the object passed to it matches the conditions defined in the delegate. The elements of the current [List<T>](#) are individually passed to the [Predicate<T>](#) delegate, and processing is stopped when the delegate returns `false` for any element. The elements are processed in order, and all calls are made on a single thread.

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Exists\(Predicate<T>\)](#)
- [Predicate<T>](#)

List<T>.ICollection<T>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool System.Collections.Generic.ICollection<T>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [List<T>](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IEnumerable<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerator<T> IEnumerable<T>.GetEnumerator();
```

Returns

[IEnumerator<T>](#)

An [IEnumerator<T>](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

List<T>.ICollection.CopyTo(Array, Int32)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int arrayIndex);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

arrayIndex [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than 0.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `arrayIndex` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the nongeneric implementations of [ICollection.CopyTo](#) throw [InvalidOperationException](#), whereas the generic implementations throw [ArgumentException](#).

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [List<T>](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. In the rare case where enumeration contends with write accesses, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#).

Synchronization is effective only if all threads lock this object before accessing the collection.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection.SyncRoot](#)

List<T>.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [List<T>](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#).

Synchronization is effective only if all threads lock this object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;
lock (ic.SyncRoot)
```

```
{  
    // Access the collection.  
}
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection.IsSynchronized](#)

List<T>.IEnumarable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumarable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator](#)

List<T>.IList.Add(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to the [IList](#).

C#

```
int IList.Add(object item);
```

Parameters

item [Object](#)

The [Object](#) to add to the [IList](#).

Returns

[Int32](#)

The position into which the new element was inserted.

Implements

[Add\(Object\)](#)

Exceptions

[ArgumentException](#)

item is of a type that is not assignable to the [IList](#).

Remarks

If [Count](#) is less than [Capacity](#), this method is an O(1) operation. If the capacity needs to be increased to accommodate the new element, this method becomes an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IList.Contains(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [IList](#) contains a specific value.

C#

```
bool IList.Contains(object item);
```

Parameters

item [Object](#)

The [Object](#) to locate in the [IList](#).

Returns

[Boolean](#)

`true` if `item` is found in the [IList](#); otherwise, `false`.

Implements

[Contains\(Object\)](#)

Remarks

This method determines equality using the default equality comparer [EqualityComparer<T>.Default](#) for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IList.IndexOf(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines the index of a specific item in the [IList](#).

C#

```
int IList.IndexOf(object item);
```

Parameters

item [Object](#)

The object to locate in the [IList](#).

Returns

[Int32](#)

The index of **item** if found in the list; otherwise, -1.

Implements

[IndexOf\(Object\)](#)

Exceptions

[ArgumentException](#)

item is of a type that is not assignable to the [IList](#).

Remarks

This method determines equality using the default equality comparer

[EqualityComparer<T>.Default](#) for **T**, the type of values in the list.

This method performs a linear search; therefore, this method is an O(n) operation, where n is

[Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IList.Insert(Int32, Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Inserts an item to the [IList](#) at the specified index.

C#

```
void IList.Insert(int index, object item);
```

Parameters

index Int32

The zero-based index at which **item** should be inserted.

item Object

The object to insert into the [IList](#).

Implements

[Insert\(Int32, Object\)](#)

Exceptions

[ArgumentOutOfRangeException](#)

index is not a valid index in the [IList](#).

[ArgumentException](#)

item is of a type that is not assignable to the [IList](#).

Remarks

If **index** equals the number of items in the [IList](#), then **item** is appended to the end.

This method is an O(*n*) operation, where *n* is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IList.IsFixedSize Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [IList](#) has a fixed size.

C#

```
bool System.Collections.IList.IsFixedSize { get; }
```

Property Value

[Boolean](#)

`true` if the [IList](#) has a fixed size; otherwise, `false`. In the default implementation of [List<T>](#), this property always returns `false`.

Implements

[IsFixedSize](#)

Remarks

A collection with a fixed size does not allow the addition or removal of elements after the collection is created, but it allows the modification of existing elements.

A collection with a fixed size is simply a collection with a wrapper that prevents adding and removing elements; therefore, if changes are made to the underlying collection, including the addition or removal of elements, the fixed-size collection reflects those changes.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IList.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [IList](#) is read-only.

C#

```
bool System.Collections.IList.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [IList](#) is read-only; otherwise, `false`. In the default implementation of [List<T>](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IList.Item[Int32] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the element at the specified index.

C#

```
object? System.Collections.IList.Item[int index] { get; set; }
```

Parameters

index [Int32](#)

The zero-based index of the element to get or set.

Property Value

[Object](#)

The element at the specified index.

Implements

[Item\[Int32\]](#)

Exceptions

[ArgumentOutOfRangeException](#)

index is not a valid index in the [IList](#).

[ArgumentException](#)

The property is set and the value is of a type that isn't assignable to the [IList](#).

Remarks

The C# language uses the [this](#) keyword to define the indexers instead of implementing the [IList.Item\[\]](#) property. Visual Basic implements [IList.Item\[\]](#) as a default property, which provides the same indexing functionality.

Retrieving the value of this property is an O(1) operation; setting the property is also an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

List<T>.IList.Remove(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a specific object from the [IList](#).

C#

```
void IList.Remove(object item);
```

Parameters

item [Object](#)

The object to remove from the [IList](#).

Implements

[Remove\(Object\)](#)

Exceptions

[ArgumentException](#)

item is of a type that is not assignable to the [IList](#).

Remarks

This method determines equality using the default equality comparer

[EqualityComparer<T>.Default](#) for **T**, the type of values in the list.

This method performs a linear search; therefore, this method is an O(*n*) operation, where *n* is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Enumerates the element and priority pairs of a [PriorityQueue<TElement,TPriority>](#), without any ordering guarantees.

C#

```
public struct  
PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator :  
System.Collections.Generic.IEnumerator<(TElement, TPriority)>
```

Type Parameters

TElement

TPriority

Inheritance [Object](#) → [ValueType](#) →
[PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator](#)

Implements [IEnumerator<ValueTuple<TElement,TPriority>>](#) , [IEnumerator](#) , [IDisposable](#)

Properties

[] [Expand table](#)

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

[] Expand table

Dispose()	Releases all resources used by the PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator .
MoveNext()	Advances the enumerator to the next element of the UnorderedItems .

Explicit Interface Implementations

[] Expand table

IEnumerator.Current	Gets the element in the collection at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public(TElement Element, TPriority Priority) Current { get; }
```

Property Value

[ValueTuple<TElement,TPriority>](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Releases all resources used by the

[PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Advances the enumerator to the next element of the [UnorderedItems](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets the element in the collection at the current position of the enumerator.

C#

```
object System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator](#) instance is cast to an [IEnumerator](#) interface.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumerator.Reset();
```

Implements

[Reset\(\)](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator](#) instance is cast to an [IEnumerator](#) interface.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Priority Queue<TElement,TPriority>.UnorderedItemsCollection Class

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Enumerates the contents of a [PriorityQueue<TElement,TPriority>](#), without any ordering guarantees.

C#

```
public sealed class PriorityQueue<TElement,TPriority>.UnorderedItemsCollection :  
    System.Collections.Generic.IEnumerable<(TElement, TPriority)>,  
    System.Collections.Generic.IReadOnlyCollection<(TElement, TPriority)>,  
    System.Collections.ICollection
```

Type Parameters

TElement

TPriority

Inheritance [Object](#) → PriorityQueue<TElement,TPriority>.UnorderedItemsCollection

Implements [IEnumerable<ValueTuple<TElement,TPriority>>](#) , [IEnumerable<T>](#) ,
[IReadOnlyCollection<ValueTuple<TElement,TPriority>>](#) , [ICollection](#) , [IEnumerable](#)

Properties

 [Expand table](#)

Count	Gets the number of elements in the collection.
-----------------------	--

Methods

[+] Expand table

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the UnorderedItems .
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)
ToString()	Returns a string that represents the current object. (Inherited from Object)

Explicit Interface Implementations

[+] Expand table

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array , starting at a particular Array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through a collection.

Extension Methods

[+] Expand table

TolImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
TolImmutableDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.

<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.

<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.

<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable <code>Int32</code> values that are obtained by invoking a

	transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.

DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.

<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.

<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.

<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter,TInner,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter,TInner,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.

<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Decimal</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Double</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int64</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Decimal</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Double</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum nullable Int64 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the maximum Single value.
Max<TSource>(IEnumerable<TSource>)	Returns the maximum value in a generic sequence.
Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function.
Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.

<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> ,

<code>Func<TSource, TCollection, TResult>()</code>	flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified <code>IEqualityComparer<T></code> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
<code>Single<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
<code>Single<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than

	<p>one element satisfies the condition.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>)	<p>Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.</p>
Skip<TSource>(IEnumerable<TSource>, Int32)	<p>Bypasses a specified number of elements in a sequence and then returns the remaining elements.</p>
SkipLast<TSource>(IEnumerable<TSource>, Int32)	<p>Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.</p>
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.</p>
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	<p>Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.</p>
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	<p>Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.</p>
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	<p>Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.</p>
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	<p>Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.</p>

Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.
Take<TSource>(IEnumerable<TSource>, Range)	Returns a specified range of contiguous elements from a

	sequence.
TakeLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns elements from a sequence as long as a specified condition is true.
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a <code>IEnumerable<T></code> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
ToList<TSource>(IEnumerable<TSource>)	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .

ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an IEnumerable to an IQueryable .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of every node in the source collection.
<code>DescendantNodes<T>(IEnumerable<T>)</code>	Returns a collection of the descendant nodes of every document and element in the source collection.
<code>Descendants<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
<code>Descendants<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the descendant

	elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Priority Queue<TElement,TPriority>.Unordered ItemsCollection.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets the number of elements in the collection.

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements in the collection.

Implements

[Count](#) , [Count](#)

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Returns an enumerator that iterates through the [UnorderedItems](#).

C#

```
public  
System.Collections.Generic.PriorityQueue<TElement,TPriority>.UnorderedItemsCollect  
ion Enumerator GetEnumerator();
```

Returns

[PriorityQueue<TElement,TPriority>.UnorderedItemsCollection](#).[Enumerator](#)

An [PriorityQueue<TElement,TPriority>.UnorderedItemsCollection](#).[Enumerator](#) for the [UnorderedItems](#).

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Priority Queue<TElement,TPriority>.Unordered ItemsCollection.ICollection.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`.

Implements

[IsSynchronized](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [PriorityQueue<TElement,TPriority>.UnorderedItemsCollection](#) instance is cast to an [ICollection](#) interface.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#).

Implements

[SyncRoot](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [PriorityQueue<TElement,TPriority>.UnorderedItemsCollection](#) instance is cast to an [ICollection](#) interface.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerable IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) object that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [PriorityQueue<TElement,TPriority>.UnorderedItemsCollection](#) instance is cast to an [IEnumerable](#) interface.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority> Class

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Represents a collection of items that have a value and a priority. On dequeue, the item with the lowest priority value is removed.

C#

```
public class PriorityQueue<TElement,TPriority>
```

Type Parameters

TElement

Specifies the type of elements in the queue.

TPriority

Specifies the type of priority associated with enqueued elements.

Inheritance [Object](#) → PriorityQueue<TElement,TPriority>

Remarks

Implements an array-backed, quaternary min-heap. Each element is enqueued with an associated priority that determines the dequeue order. Elements with the lowest priority are dequeued first. Note that the type does not guarantee first-in-first-out semantics for elements of equal priority.

Constructors

 [Expand table](#)

PriorityQueue<TElement,TPriority>()	Initializes a new instance of the PriorityQueue<TElement,TPriority> class.
PriorityQueue<TElement,TPriority>(IComparer<TPriority>)	Initializes a new instance of the PriorityQueue<TElement,TPriority> class with the

	specified custom priority comparer.
<code>PriorityQueue<TElement,TPriority>(IEnumerable<ValueTuple<TElement,TPriority>>, IComparer<TPriority>)</code>	Initializes a new instance of the <code>PriorityQueue<TElement,TPriority></code> class that is populated with the specified elements and priorities, and with the specified custom priority comparer.
<code>PriorityQueue<TElement,TPriority>(IEnumerable<ValueTuple<TElement,TPriority>>)</code>	Initializes a new instance of the <code>PriorityQueue<TElement,TPriority></code> class that is populated with the specified elements and priorities.
<code>PriorityQueue<TElement,TPriority>(Int32, IComparer<TPriority>)</code>	Initializes a new instance of the <code>PriorityQueue<TElement,TPriority></code> class with the specified initial capacity and custom priority comparer.
<code>PriorityQueue<TElement,TPriority>(Int32)</code>	Initializes a new instance of the <code>PriorityQueue<TElement,TPriority></code> class with the specified initial capacity.

Properties

[] [Expand table](#)

<code>Comparer</code>	Gets the priority comparer used by the <code>PriorityQueue<TElement,TPriority></code> .
<code>Count</code>	Gets the number of elements contained in the <code>PriorityQueue<TElement,TPriority></code> .
<code>UnorderedItems</code>	Gets a collection that enumerates the elements of the queue in an unordered manner.

Methods

[] [Expand table](#)

<code>Clear()</code>	Removes all items from the <code>PriorityQueue<TElement,TPriority></code> .
<code>Dequeue()</code>	Removes and returns the minimal element from the <code>PriorityQueue<TElement,TPriority></code> - that is, the element with the lowest priority value.
<code>Enqueue(TElement, TPriority)</code>	Adds the specified element with associated priority to the <code>PriorityQueue<TElement,TPriority></code> .
<code>EnqueueDequeue(TElement, TPriority)</code>	Adds the specified element with associated priority to the <code>PriorityQueue<TElement,TPriority></code> , and immediately removes

	the minimal element, returning the result.
Enqueue Range(IEnumerable<TElement>, TPriority)	Enqueues a sequence of elements pairs to the PriorityQueue<TElement,TPriority> , all associated with the specified priority.
EnqueueRange(IEnumerable<Value Tuple<TElement,TPriority>>)	Enqueues a sequence of element-priority pairs to the PriorityQueue<TElement,TPriority> .
EnsureCapacity(Int32)	Ensures that the PriorityQueue<TElement,TPriority> can hold up to <code>capacity</code> items without further expansion of its backing storage.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
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 minimal element from the PriorityQueue<TElement,TPriority> without removing it.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity to the actual number of items in the PriorityQueue<TElement,TPriority> , if that is less than 90 percent of current capacity.
TryDequeue(TElement, TPriority)	Removes the minimal element from the PriorityQueue<TElement,TPriority> , and copies it and its associated priority to the <code>element</code> and <code>priority</code> arguments.
TryPeek(TElement, TPriority)	Returns a value that indicates whether there is a minimal element in the PriorityQueue<TElement,TPriority> , and if one is present, copies it and its associated priority to the <code>element</code> and <code>priority</code> arguments. The element is not removed from the PriorityQueue<TElement,TPriority> .

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>

Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Overloads

 [Expand table](#)

PriorityQueue<TElement,TPriority>()	Initializes a new instance of the PriorityQueue<TElement,TPriority> class.
PriorityQueue<TElement,TPriority>(IComparer<TPriority>)	Initializes a new instance of the PriorityQueue<TElement,TPriority> class with the specified custom priority comparer.
PriorityQueue<TElement,TPriority>(IEnumerable<ValueTuple<TElement,TPriority>>)	Initializes a new instance of the PriorityQueue<TElement,TPriority> class that is populated with the specified elements and priorities.
PriorityQueue<TElement,TPriority>(Int32)	Initializes a new instance of the PriorityQueue<TElement,TPriority> class with the specified initial capacity.
PriorityQueue<TElement,TPriority>(IEnumerable<ValueTuple<TElement,TPriority>>, IComparer<TPriority>)	Initializes a new instance of the PriorityQueue<TElement,TPriority> class that is populated with the specified elements and priorities, and with the specified custom priority comparer.
PriorityQueue<TElement,TPriority>(Int32, IComparer<TPriority>)	Initializes a new instance of the PriorityQueue<TElement,TPriority> class with the specified initial capacity and custom priority comparer.

PriorityQueue<TElement,TPriority>()

Initializes a new instance of the [PriorityQueue<TElement,TPriority>](#) class.

C#

```
public PriorityQueue();
```

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority> (IComparer<TPriority>)

Initializes a new instance of the [PriorityQueue<TElement,TPriority>](#) class with the specified custom priority comparer.

C#

```
public PriorityQueue(System.Collections.Generic.IComparer<TPriority>?
comparer);
```

Parameters

comparer [IComparer<TPriority>](#)

Custom comparer dictating the ordering of elements. Uses [Default](#) if the argument is [null](#).

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority> (IEnumerable<ValueTuple<TElement,TPriority>>)

Initializes a new instance of the `PriorityQueue<TElement,TPriority>` class that is populated with the specified elements and priorities.

C#

```
public PriorityQueue(System.Collections.Generic.IEnumerable<(TElement Element,  
TPriority Priority)> items);
```

Parameters

items `IEnumerable<ValueTuple<TElement,TPriority>>`

The pairs of elements and priorities with which to populate the queue.

Exceptions

`ArgumentNullException`

The specified `items` argument was `null`.

Remarks

Constructs the heap using a heapify operation, which is generally faster than enqueueing individual elements sequentially.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>(Int32)

Initializes a new instance of the `PriorityQueue<TElement,TPriority>` class with the specified initial capacity.

C#

```
public PriorityQueue(int initialCapacity);
```

Parameters

initialCapacity `Int32`

Initial capacity to allocate in the underlying heap array.

Exceptions

[ArgumentOutOfRangeException](#)

The specified `initialCapacity` was negative.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority> ([IEnumerable<ValueTuple<TElement,TPriority>>](#), [IComparer<TPriority>](#))

Initializes a new instance of the [PriorityQueue<TElement,TPriority>](#) class that is populated with the specified elements and priorities, and with the specified custom priority comparer.

C#

```
public PriorityQueue(System.Collections.Generic.IEnumerable<(TElement Element,
TPriority Priority)> items, System.Collections.Generic.IComparer<TPriority>?
comparer);
```

Parameters

items `IEnumerable<ValueTuple<TElement,TPriority>>`

The pairs of elements and priorities with which to populate the queue.

comparer `IComparer<TPriority>`

Custom comparer dictating the ordering of elements. Uses [Default](#) if the argument is `null`.

Exceptions

ArgumentNullException

The specified `items` argument was `null`.

Remarks

Constructs the heap using a heapify operation, which is generally faster than enqueueing individual elements sequentially.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>(Int32, IComparer<TPriority>)

Initializes a new instance of the `PriorityQueue<TElement,TPriority>` class with the specified initial capacity and custom priority comparer.

C#

```
public PriorityQueue(int initialCapacity,  
System.Collections.Generic.IComparer<TPriority>? comparer);
```

Parameters

`initialCapacity` `Int32`

Initial capacity to allocate in the underlying heap array.

`comparer` `IComparer<TPriority>`

Custom comparer dictating the ordering of elements. Uses `Default` if the argument is `null`.

Exceptions

ArgumentOutOfRangeException

The specified `initialCapacity` was negative.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.Comparer Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets the priority comparer used by the [PriorityQueue<TElement,TPriority>](#).

C#

```
public System.Collections.Generic.IComparer<TPriority> Comparer { get; }
```

Property Value

[IComparer<TPriority>](#)

The comparer that determines the priority of items in the queue.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets the number of elements contained in the [PriorityQueue<TElement,TPriority>](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the queue.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.UnorderedItems Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets a collection that enumerates the elements of the queue in an unordered manner.

C#

```
public  
System.Collections.Generic.PriorityQueue<TElement,TPriority>.UnorderedItemsCollect  
ion UnorderedItems { get; }
```

Property Value

[PriorityQueue<TElement,TPriority>.UnorderedItemsCollection](#)

Remarks

The enumeration does not order items by priority, since that would require $N * \log(N)$ time and N space. Instead, items are enumerated following the internal array heap layout.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Removes all items from the [PriorityQueue<TElement,TPriority>](#).

C#

```
public void Clear();
```

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Priority Queue<TElement,TPriority>.Dequeue Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Removes and returns the minimal element from the [PriorityQueue<TElement,TPriority>](#) - that is, the element with the lowest priority value.

C#

```
public TElement Dequeue();
```

Returns

TElement

The minimal element of the [PriorityQueue<TElement,TPriority>](#).

Exceptions

[InvalidOperationException](#)

The queue is empty.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Priority Queue<TElement,TPriority>.Enqueue(TElement, TPriority) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Adds the specified element with associated priority to the [PriorityQueue<TElement,TPriority>](#).

C#

```
public void Enqueue(TElement element, TPriority priority);
```

Parameters

element TElement

The element to add to the [PriorityQueue<TElement,TPriority>](#).

priority TPriority

The priority with which to associate the new element.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Priority Queue<TElement,TPriority>.Enqueue Dequeue Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Adds the specified element with associated priority to the [PriorityQueue<TElement,TPriority>](#), and immediately removes the minimal element, returning the result.

C#

```
public TElement EnqueueDequeue(TElement element, TPriority priority);
```

Parameters

element TElement

The element to add to the [PriorityQueue<TElement,TPriority>](#).

priority TPriority

The priority with which to associate the new element.

Returns

TElement

The minimal element removed after the enqueue operation.

Remarks

Implements an insert-then-extract heap operation that's generally more efficient than sequencing enqueue and dequeue operations.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Priority Queue<TElement,TPriority>.EnqueueRange Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Overloads

[+] Expand table

EnqueueRange(IEnumerable<ValueTuple<TElement,TPriority>>)	Enqueues a sequence of element-priority pairs to the PriorityQueue<TElement,TPriority> .
EnqueueRange(IEnumerable<TElement>, TPriority)	Enqueues a sequence of elements pairs to the PriorityQueue<TElement,TPriority> , all associated with the specified priority.

EnqueueRange(IEnumerable<ValueTuple<TElement,TPriority>>)

Enqueues a sequence of element-priority pairs to the [PriorityQueue<TElement,TPriority>](#).

C#

```
public void EnqueueRange(System.Collections.Generic.IEnumerable<(TElement Element, TPriority Priority)> items);
```

Parameters

items [IEnumerable<ValueTuple<TElement,TPriority>>](#)

The pairs of elements and priorities to add to the queue.

Exceptions

ArgumentNullException

The specified `items` argument was `null`.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

EnqueueRange(IEnumerable<TElement>, TPriority)

Enqueues a sequence of elements pairs to the [PriorityQueue<TElement,TPriority>](#), all associated with the specified priority.

C#

```
public void EnqueueRange(System.Collections.Generic.IEnumerable<TElement>
elements, TPriority priority);
```

Parameters

elements `IEnumerable<TElement>`

The elements to add to the queue.

priority `TPriority`

The priority to associate with the new elements.

Exceptions

ArgumentNullException

The specified `elements` argument was `null`.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.EnsureCapacity(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Ensures that the [PriorityQueue<TElement,TPriority>](#) can hold up to `capacity` items without further expansion of its backing storage.

C#

```
public int EnsureCapacity(int capacity);
```

Parameters

capacity [Int32](#)

The minimum capacity to be used.

Returns

[Int32](#)

The current capacity of the [PriorityQueue<TElement,TPriority>](#).

Exceptions

[ArgumentOutOfRangeException](#)

The specified `capacity` is negative.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.Peek Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Returns the minimal element from the [PriorityQueue<TElement,TPriority>](#) without removing it.

C#

```
public TElement Peek();
```

Returns

TElement

The minimal element of the [PriorityQueue<TElement,TPriority>](#).

Exceptions

[InvalidOperationException](#)

The [PriorityQueue<TElement,TPriority>](#) is empty.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.TrimExcess Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Sets the capacity to the actual number of items in the [PriorityQueue<TElement,TPriority>](#), if that is less than 90 percent of current capacity.

C#

```
public void TrimExcess();
```

Remarks

This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.TryDequeue Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Removes the minimal element from the [PriorityQueue<TElement,TPriority>](#), and copies it and its associated priority to the `element` and `priority` arguments.

C#

```
public bool TryDequeue(out TElement element, out TPriority priority);
```

Parameters

element [TElement](#)

When this method returns, contains the removed element.

priority [TPriority](#)

When this method returns, contains the priority associated with the removed element.

Returns

[Boolean](#)

`true` if the element is successfully removed; `false` if the [PriorityQueue<TElement,TPriority>](#) is empty.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

PriorityQueue<TElement,TPriority>.TryPeek(TElement, TPriority) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Returns a value that indicates whether there is a minimal element in the [PriorityQueue<TElement,TPriority>](#), and if one is present, copies it and its associated priority to the `element` and `priority` arguments. The element is not removed from the [PriorityQueue<TElement,TPriority>](#).

C#

```
public bool TryPeek(out TElement element, out TPriority priority);
```

Parameters

element [TElement](#)

When this method returns, contains the minimal element in the queue.

priority [TPriority](#)

When this method returns, contains the priority associated with the minimal element.

Returns

[Boolean](#)

`true` if there is a minimal element; `false` if the [PriorityQueue<TElement,TPriority>](#) is empty.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Queue<T>.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [Queue<T>](#).

C#

```
public struct Queue<T>.Enumerator : System.Collections.Generic.IEnumerator<T>
```

Type Parameters

T

Inheritance [Object](#) → [ValueType](#) → Queue<T>.Enumerator

Implements [IEnumerator<T>](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned

`false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to `MoveNext` or `IEnumerator.Reset` throws an `InvalidOperationException`.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in `System.Collections.Generic` are not synchronized.

Properties

[+] Expand table

<code>Current</code>	Gets the element at the current position of the enumerator.
----------------------	---

Methods

[+] Expand table

<code>Dispose()</code>	Releases all resources used by the <code>Queue<T>.Enumerator</code> .
<code>MoveNext()</code>	Advances the enumerator to the next element of the <code>Queue<T></code> .

Explicit Interface Implementations

[+] Expand table

<code>IEnumerator.Current</code>	Gets the element at the current position of the enumerator.
<code>IEnumerator.Reset()</code>	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

Queue<T>.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the [Queue<T>](#) at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

Queue<T>.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [Queue<T>.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Queue<T>.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [Queue<T>](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to [MoveNext](#) advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is

irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

Queue<T>.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

`IEnumerator.Current` does not move the position of the enumerator, and consecutive calls to `IEnumerator.Current` return the same object until either `MoveNext` or `IEnumerator.Reset` is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator.Reset\(\)](#)

Queue<T>.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator.Current](#)
- [Current](#)

Queue<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a first-in, first-out collection of objects.

C#

```
public class Queue<T> : System.Collections.Generic.IEnumerable<T>,
System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.ICollection
```

Type Parameters

T

Specifies the type of elements in the queue.

Inheritance [Object](#) → Queue<T>

Implements [IEnumerable<T>](#) , [IReadOnlyCollection<T>](#) , [ICollection](#) , [IEnumerable](#)

Examples

The following code example demonstrates several methods of the [Queue<T>](#) 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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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
Dequeueing 'two'

Contents of the first copy:
three
four
five

Contents of the second copy, with duplicates and nulls:

three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

This class implements a generic queue as a circular array. Objects stored in a `Queue<T>` 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<T>](#) if you need to access the information in the same order that it is stored in the collection. Use [Stack<T>](#) if you need to access the information in reverse order. Use [ConcurrentQueue<T>](#) or [ConcurrentStack<T>](#) if you need to access the collection from multiple threads concurrently.

Three main operations can be performed on a [Queue<T>](#) and its elements:

- [Enqueue](#) adds an element to the end of the [Queue<T>](#).
- [Dequeue](#) removes the oldest element from the start of the [Queue<T>](#).
- [Peek](#) peek returns the oldest element that is at the start of the [Queue<T>](#) but does not remove it from the [Queue<T>](#).

The capacity of a [Queue<T>](#) is the number of elements the [Queue<T>](#) can hold. As elements are added to a [Queue<T>](#), the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling [TrimExcess](#).

[Queue<T>](#) accepts `null` as a valid value for reference types and allows duplicate elements.

Constructors

[] [Expand table](#)

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

[] [Expand table](#)

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

Methods

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

Explicit Interface Implementations

[Expand table](#)

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array , starting at a particular Array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through a collection.
IEnumerable<T>.Get Enumerator()	Returns an enumerator that iterates through a collection.

Extension Methods

[Expand table](#)

TolImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
TolImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an

	immutable dictionary of its contents.
ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
ToImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.
ToImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .

CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource>)	Applies an accumulator function over a sequence.
Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether all elements of a sequence satisfy a condition.
Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether any element of a sequence satisfies a condition.
Any<TSource>(IEnumerable<TSource>)	Determines whether a sequence contains any elements.
Append<TSource>(IEnumerable<TSource>, TSource)	Appends a value to the end of the sequence.
AsEnumerable<TSource>(IEnumerable<TSource>)	Returns the input typed as IEnumerable<T> .
Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the average of a sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the average of a sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the average of a sequence of Int32 values that are obtained by invoking a transform

	function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.

<code>Chunk<TSource>(IEnumerable<TSource>, Int32)</code>	Splits the elements of a sequence into chunks of size at most <code>size</code> .
<code>Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Concatenates two sequences.
<code>Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)</code>	Determines whether a sequence contains a specified element by using a specified <code>IEqualityComparer<T></code> .
<code>Contains<TSource>(IEnumerable<TSource>, TSource)</code>	Determines whether a sequence contains a specified element by using the default equality comparer.
<code>Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns a number that represents how many elements in the specified sequence satisfy a condition.
<code>Count<TSource>(IEnumerable<TSource>)</code>	Returns the number of elements in a sequence.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
<code>DefaultIfEmpty<TSource>(IEnumerable<TSource>)</code>	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.

<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition,

	or a specified default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence, or a default value if the sequence contains no elements.
GroupBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
GroupBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Groups the elements of a sequence according to a specified key selector function.
GroupBy<TSource, TKey, TElement>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource, TElement>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
GroupBy<TSource, TKey, TElement>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource, TElement>)	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
GroupBy<TSource, TKey, TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TKey, IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
GroupBy<TSource, TKey, TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TKey, IEnumerable<TSource>, TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
GroupBy<TSource, TKey, TElement, TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource, TElement>, TResult)	Groups the elements of a sequence according to a specified

<code>Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>()</code>	key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.

<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.

<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int64</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Decimal</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Double</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Int64</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Single</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Single</code> value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.

<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.

Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)	Projects each element of a sequence into a new form by incorporating the element's index.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)	Projects each element of a sequence to an IEnumerable<T> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.

<code>Single<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>Skip<TSource>(IEnumerable<TSource>, Int32)</code>	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
<code>SkipLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
<code>SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.

SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code>

	according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.

<code>Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set union of two sequences by using a specified <code>IEqualityComparer<T></code> .
<code>Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set union of two sequences by using the default equality comparer.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)</code>	Produces the set union of two sequences according to a specified key selector function.
<code>Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Filters a sequence of values based on a predicate.
<code>Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an <code>IEnumerable</code> to an <code>IQueryable</code> .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic <code>IEnumerable<T></code> to a generic <code>IQueryable<T></code> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements

	that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`Shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [Queue<T>](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. For a thread-safe queue, see [ConcurrentQueue<T>](#).

Queue<T> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [Queue<T>](#) class.

Overloads

[+] [Expand table](#)

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.

Examples

The following code example demonstrates this constructor and several other methods of the [Queue<T>](#) 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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

```
C#  
  
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

Contents of the second copy, with duplicates and nulls:

three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Queue<T>()

Initializes a new instance of the [Queue<T>](#) class that is empty and has the default initial capacity.

C#

```
public Queue();
```

Remarks

The capacity of a [Queue<T>](#) is the number of elements that the [Queue<T>](#) can hold. As elements are added to a [Queue<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Queue<T>](#).

The capacity can be decreased by calling [TrimExcess](#).

This constructor is an O(1) operation.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

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.

C#

```
public Queue(System.Collections.Generic.IEnumerable<T> collection);
```

Parameters

collection [IEnumerable<T>](#)

The collection whose elements are copied to the new [Queue<T>](#).

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Remarks

The capacity of a [Queue<T>](#) is the number of elements that the [Queue<T>](#) can hold. As elements are added to a [Queue<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Queue<T>](#).

The capacity can be decreased by calling [TrimExcess](#).

The elements are copied onto the [Queue<T>](#) in the same order they are read by the [IEnumerator<T>](#) of the collection.

This constructor is an $O(n)$ operation, where n is the number of elements in `collection`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

[Queue<T>\(Int32\)](#)

Initializes a new instance of the [Queue<T>](#) class that is empty and has the specified initial capacity.

C#

```
public Queue(int capacity);
```

Parameters

capacity Int32

The initial number of elements that the [Queue<T>](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#)

`capacity` is less than zero.

Remarks

The capacity of a [Queue<T>](#) is the number of elements that the [Queue<T>](#) can hold. As elements are added to a [Queue<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Queue<T>](#).

The capacity can be decreased by calling [TrimExcess](#).

This constructor is an O(`n`) operation, where `n` is `capacity`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Queue<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements contained in the [Queue<T>](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [Queue<T>](#).

Implements

[Count](#), [Count](#)

Examples

The following code example demonstrates several properties and methods of the [Queue<T>](#) generic class, including the [Count](#) property.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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

```

Contents of the second copy, with duplicates and nulls:

```

three
four
five

```

```

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

The capacity of a [Queue<T>](#) is the number of elements that the [Queue<T>](#) can store. [Count](#) is the number of elements that are actually in the [Queue<T>](#).

The capacity is always greater than or equal to [Count](#). If [Count](#) exceeds the capacity while adding elements, the capacity is increased by automatically reallocating the internal array before copying the old elements and adding the new elements.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Queue<T>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all objects from the [Queue<T>](#).

C#

```
public void Clear();
```

Examples

The following code example demonstrates several methods of the [Queue<T>](#) generic class, including the [Clear](#) method.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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
```

```
Contents of the second copy, with duplicates and nulls:
```

```
three  
four  
five
```

```
queueCopy.Contains("four") = True  
  
queueCopy.Clear()  
  
queueCopy.Count = 0  
*/
```

Remarks

`Count` is set to zero, and references to other objects from elements of the collection are also released.

The capacity remains unchanged. To reset the capacity of the `Queue<T>`, call `TrimExcess`. Trimming an empty `Queue<T>` sets the capacity of the `Queue<T>` to the default capacity.

This method is an $O(n)$ operation, where n is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [TrimExcess\(\)](#)

Queue<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether an element is in the [Queue<T>](#).

C#

```
public bool Contains(T item);
```

Parameters

item [T](#)

The object to locate in the [Queue<T>](#). The value can be `null` for reference types.

Returns

[Boolean](#)

`true` if `item` is found in the [Queue<T>](#); otherwise, `false`.

Examples

The following code example demonstrates several methods of the [Queue<T>](#) generic class, including the [Contains](#) method.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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

Contents of the second copy, with duplicates and nulls:

three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

This method determines equality using the default equality comparer `EqualityComparer<T>.Default` for `T`, the type of values in the queue.

This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Performing Culture-Insensitive String Operations](#)

Queue<T>.CopyTo(T[], Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [Queue<T>](#) elements to an existing one-dimensional [Array](#), starting at the specified array index.

C#

```
public void CopyTo(T[] array, int arrayIndex);
```

Parameters

array [T\[\]](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [Queue<T>](#). The [Array](#) must have zero-based indexing.

arrayIndex [Int32](#)

The zero-based index in [array](#) at which copying begins.

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than zero.

[ArgumentException](#)

The number of elements in the source [Queue<T>](#) is greater than the available space from [arrayIndex](#) to the end of the destination [array](#).

Remarks

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [Queue<T>](#).

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Queue<T>.Dequeue Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes and returns the object at the beginning of the [Queue<T>](#).

C#

```
public T Dequeue();
```

Returns

T

The object that is removed from the beginning of the [Queue<T>](#).

Exceptions

[InvalidOperationException](#)

The [Queue<T>](#) is empty.

Examples

The following code example demonstrates several methods of the [Queue<T>](#) generic class, including the [Dequeue](#) method.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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

Contents of the second copy, with duplicates and nulls:

three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

This method is similar to the [Peek](#) method, but [Peek](#) does not modify the [Queue<T>](#).

If type `T` is a reference type, `null` can be added to the [Queue<T>](#) as a value.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Enqueue\(T\)](#)
- [Peek\(\)](#)

Queue<T>.Enqueue(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an object to the end of the [Queue<T>](#).

C#

```
public void Enqueue(T item);
```

Parameters

item T

The object to add to the [Queue<T>](#). The value can be `null` for reference types.

Examples

The following code example demonstrates several methods of the [Queue<T>](#) generic class, including the [Enqueue](#) method.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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

Contents of the second copy, with duplicates and nulls:

three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

If [Count](#) already equals the capacity, the capacity of the [Queue<T>](#) is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If [Count](#) is less than the capacity of the internal array, this method is an O(1) operation. If the internal array needs to be reallocated to accommodate the new element, this method becomes an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Dequeue\(\)](#)
- [Peek\(\)](#)

Queue<T>.EnsureCapacity(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

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`.

C#

```
public int EnsureCapacity(int capacity);
```

Parameters

capacity [Int32](#)

The minimum capacity to ensure.

Returns

[Int32](#)

The new capacity of this queue.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Queue<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [Queue<T>](#).

C#

```
public System.Collections.Generic.Queue<T>.Enumerator GetEnumerator();
```

Returns

[Queue<T>.Enumerator](#)

An [Queue<T>.Enumerator](#) for the [Queue<T>](#).

Examples

The following code example demonstrates that the [Queue<T>](#) generic class is enumerable. The `foreach` statement (`For Each` in Visual Basic) is used to enumerate the queue.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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

Contents of the second copy, with duplicates and nulls:

three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Queue<T>.Enumerator](#)
- [IEnumerator<T>](#)

Queue<T>.Peek Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns the object at the beginning of the [Queue<T>](#) without removing it.

C#

```
public T Peek();
```

Returns

T

The object at the beginning of the [Queue<T>](#).

Exceptions

[InvalidOperationException](#)

The [Queue<T>](#) is empty.

Examples

The following code example demonstrates several methods of the [Queue<T>](#) generic class, including the [Peek](#) method.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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
Dequeueing 'two'

```

```

Contents of the first copy:
three
four
five

```

Contents of the second copy, with duplicates and nulls:

```

three
four
five

```

```

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

This method is similar to the [Dequeue](#) method, but [Peek](#) does not modify the [Queue<T>](#).

If type `T` is a reference type, `null` can be added to the [Queue<T>](#) as a value.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Enqueue\(T\)](#)
- [Dequeue\(\)](#)

Queue<T>.ToArray Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [Queue<T>](#) elements to a new array.

C#

```
public T[] ToArray();
```

Returns

T[]

A new array containing elements copied from the [Queue<T>](#).

Examples

The following code example demonstrates several methods of the [Queue<T>](#) generic class, including the [ToArray](#) method.

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<T>](#) constructor that takes [IEnumerable<T>](#), 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<T>](#) 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.

C#

```
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

Contents of the second copy, with duplicates and nulls:

three
four
five

queueCopy.Contains("four") = True

queueCopy.Clear()

queueCopy.Count = 0
*/

```

Remarks

The [Queue<T>](#) is not modified. The order of the elements in the new array is the same as the order of the elements from the beginning of the [Queue<T>](#) to its end.

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Queue<T>.TrimExcess Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Overloads

[] [Expand table](#)

[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.

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.

C#

```
public void TrimExcess();
```

Remarks

This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large [Queue<T>](#) can be considerable, however, so the [TrimExcess](#) method does nothing if the list is at more than 90 percent of capacity. This avoids incurring a large reallocation cost for a relatively small gain.

This method is an $O(n)$ operation, where n is [Count](#).

To reset a [Queue<T>](#) to its initial state, call the [Clear](#) method before calling [TrimExcess](#) method. Trimming an empty [Queue<T>](#) sets the capacity of the [Queue<T>](#) to the default capacity.

See also

- [Clear\(\)](#)
- [Count](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Queue<T>.TryDequeue(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the object at the beginning of the [Queue<T>](#), and copies it to the `result` parameter.

C#

```
public bool TryDequeue(out T result);
```

Parameters

result `T`

The removed object.

Returns

[Boolean](#)

`true` if the object is successfully removed; `false` if the [Queue<T>](#) is empty.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Queue<T>.TryPeek(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

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>](#).

C#

```
public bool TryPeek(out T result);
```

Parameters

`result` `T`

If present, the object at the beginning of the [Queue<T>](#); otherwise, the default value of `T`.

Returns

`Boolean`

`true` if there is an object at the beginning of the [Queue<T>](#); `false` if the [Queue<T>](#) is empty.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Queue<T>.IEnumarable<T>.Get Enumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerator<T> IEnumarable<T>.GetEnumerator();
```

Returns

[IEnumerator<T>](#)

An [IEnumerator<T>](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetEnumerator\(\)](#)
- [GetInternalEnumerator\(\)](#)
- [IEnumerator<T>](#)

Queue<T>.ICollection.CopyTo(Array, Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the non-generic implementations of [ICollection.CopyTo](#) throw [InvalidOperationException](#), whereas the generic implementations throw [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Queue<T>.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [Queue<T>](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object, which can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock this object before accessing the collection.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

Queue<T>.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [Queue<T>](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object, which can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock this object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;  
lock (ic.SyncRoot) {
```

```
// Access the collection.  
}
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

Queue<T>.IEnumarable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumarable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetEnumerator\(\)](#)
- [GetInternalEnumerator\(\)](#)
- [IEnumerator](#)

ReferenceEqualityComparer Class

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

An [IEqualityComparer<T>](#) that uses reference equality ([ReferenceEquals\(Object, Object\)](#)) instead of value equality ([Equals\(Object\)](#)) when comparing two object instances.

C#

```
public sealed class ReferenceEqualityComparer :  
    System.Collections.Generic.IEqualityComparer<object>,  
    System.Collections.IEqualityComparer
```

Inheritance [Object](#) → ReferenceEqualityComparer

Implements [IEqualityComparer<Object>](#) , [IEqualityComparer](#)

Remarks

The [ReferenceEqualityComparer](#) type cannot be instantiated. Instead, use the [Instance](#) property to access the singleton instance of this type.

Properties

[] [Expand table](#)

Instance	Gets the singleton ReferenceEqualityComparer instance.
--------------------------	--

Methods

[] [Expand table](#)

Equals(Object, Object)	Determines whether two object references refer to the same object instance.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)

GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetHashCode(Object)	Returns a hash code for the specified object. The returned hash code is based on the object identity, not on the contents of the object.
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object . (Inherited from Object)
ToString()	Returns a string that represents the current object. (Inherited from Object)

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

ReferenceEqualityComparer.Instance Property

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Gets the singleton [ReferenceEqualityComparer](#) instance.

C#

```
public static System.Collections.Generic.ReferenceEqualityComparer Instance { get;  
}
```

Property Value

[ReferenceEqualityComparer](#)

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

ReferenceEqualityComparer.Equals(Object, Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Determines whether two object references refer to the same object instance.

C#

```
public bool Equals(object? x, object? y);
```

Parameters

x [Object](#)

The first object to compare.

y [Object](#)

The second object to compare.

Returns

[Boolean](#)

`true` if both `x` and `y` refer to the same object instance or if both are `null`; otherwise, `false`.

Implements

[Equals\(T, T\)](#) , [Equals\(Object, Object\)](#)

Remarks

This API is a wrapper around [ReferenceEquals\(Object, Object\)](#). It is not necessarily equivalent to calling [Equals\(Object, Object\)](#).

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

ReferenceEqualityComparer.GetHashCode(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assembly: System.Collections.dll

Returns a hash code for the specified object. The returned hash code is based on the object identity, not on the contents of the object.

C#

```
public int GetHashCode(object? obj);
```

Parameters

obj [Object](#)

The object for which to retrieve the hash code.

Returns

[Int32](#)

A hash code for the identity of `obj`.

Implements

[GetHashCode\(T\)](#) , [GetHashCode\(Object\)](#)

Remarks

This API is a wrapper around [GetHashCode\(Object\)](#). It is not necessarily equivalent to calling [GetHashCode\(\)](#).

Applies to

Product	Versions
.NET	5, 6, 7, 8, 9, 10

SortedDictionary< TKey, TValue >.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [SortedDictionary< TKey, TValue >](#).

C#

```
public struct SortedDictionary< TKey, TValue >.Enumerator :  
    System.Collections.Generic.IEnumerator< System.Collections.Generic.KeyValuePair< TKey, TValue > >, System.Collections.IDictionaryEnumerator
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [ValueType](#) → [SortedDictionary< TKey, TValue >.Enumerator](#)

Implements [IEnumerator< KeyValuePair< TKey, TValue > >](#), [IDictionaryEnumerator](#),
[IEnumerator](#), [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call the [MoveNext](#) method to advance the enumerator to the

first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Properties

[+] [Expand table](#)

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

[+] [Expand table](#)

Dispose()	Releases all resources used by the SortedDictionary<TKey,TValue>.Enumerator .
---------------------------	---

MoveNext()	Advances the enumerator to the next element of the SortedDictionary<TKey,TValue> .
----------------------------	--

Explicit Interface Implementations

IDictionaryEnumerator.Entry	Gets the element at the current position of the enumerator as a DictionaryEntry structure.
IDictionaryEnumerator.Key	Gets the key of the element at the current position of the enumerator.
IDictionaryEnumerator.Value	Gets the value of the element at the current position of the enumerator.
IEnumerator.Current	Gets the element at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [SortedDictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

SortedDictionary< TKey, TValue >.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public System.Collections.Generic.KeyValuePair< TKey, TValue > Current { get; }
```

Property Value

[KeyValuePair< TKey, TValue >](#)

The element in the [SortedDictionary< TKey, TValue >](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)

SortedDictionary< TKey, TValue >.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [SortedDictionary< TKey, TValue >.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [SortedDictionary< TKey, TValue >](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to the [MoveNext](#) method advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this

position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

SortedDictionary< TKey, TValue >.Enumerator.IDictionaryEnumerator.Entry Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator as a [DictionaryEntry](#) structure.

C#

```
System.Collections.DictionaryEntry System.Collections.IDictionaryEnumerator.Entry
{ get; }
```

Property Value

[DictionaryEntry](#)

The element in the collection at the current position of the dictionary, as a [DictionaryEntry](#) structure.

Implements

[Entry](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IDictionaryEnumerator.Entry](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The

[MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IDictionaryEnumerator.Entry](#) property.

- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IDictionaryEnumerator.Entry](#) does not move the position of the enumerator, and consecutive calls to [IDictionaryEnumerator.Entry](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

Sorted Dictionary< TKey, TValue >. Enumerator. IDictionaryEnumerator.Key Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the key of the element at the current position of the enumerator.

C#

```
object System.Collections.IDictionaryEnumerator.Key { get; }
```

Property Value

[Object](#)

The key of the element in the collection at the current position of the enumerator.

Implements

[Key](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IDictionaryEnumerator.Key](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IDictionaryEnumerator.Key](#) property.

- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IDictionaryEnumerator.Key](#) does not move the position of the enumerator, and consecutive calls to [IDictionaryEnumerator.Key](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.Enumerator.IDictionaryEnumerator.Value Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the value of the element at the current position of the enumerator.

C#

```
object? System.Collections.IDictionaryEnumerator.Value { get; }
```

Property Value

[Object](#)

The value of the element in the collection at the current position of the enumerator.

Implements

[Value](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IDictionaryEnumerator.Value](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IDictionaryEnumerator.Value](#) property.

- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IDictionaryEnumerator.Value](#) does not move the position of the enumerator, and consecutive calls to [IDictionaryEnumerator.Value](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [Current](#)
- [MoveNext\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.

- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IEnumerator.Current](#) does not move the position of the enumerator, and consecutive calls to [IEnumerator.Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumerator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling the [IEnumerator.Reset](#) method, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

SortedDictionary<TKey,TValue>.KeyCollection.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [SortedDictionary<TKey,TValue>.KeyCollection](#).

C#

```
public struct SortedDictionary<TKey, TValue>.KeyCollection.Enumerator :  
    System.Collections.Generic.IEnumerator<TKey>
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [ValueType](#) → [SortedDictionary<TKey,TValue>.KeyCollection.Enumerator](#)

Implements [IEnumerator<TKey>](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Properties

[] [Expand table](#)

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

[] [Expand table](#)

Dispose()	Releases all resources used by the SortedDictionary<TKey,TValue>.KeyCollection.Enumerator .
MoveNext()	Advances the enumerator to the next element of the SortedDictionary<TKey,TValue>.KeyCollection .

Explicit Interface Implementations

[] [Expand table](#)

IEnumerator.Current	Gets the element at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.Enumerator](#)
- [SortedDictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

SortedDictionary< TKey, TValue >.KeyCollection.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public TKey Current { get; }
```

Property Value

TKey

The element in the [SortedDictionary< TKey, TValue >.KeyCollection](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.KeyCollection.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [SortedDictionary< TKey, TValue >.KeyCollection.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue>.KeyCollection.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [SortedDictionary<TKey,TValue>.KeyCollection](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to the [MoveNext](#) method advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

SortedDictionary< TKey, TValue >.KeyCollection.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.

- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IEnumerator.Current](#) does not move the position of the enumerator, and consecutive calls to [IEnumerator.Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.KeyCollection.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumerator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling the [IEnumerator.Reset](#) method, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.Key Collection Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents the collection of keys in a [SortedDictionary< TKey, TValue >](#). This class cannot be inherited.

C#

```
public sealed class SortedDictionary< TKey, TValue >.KeyCollection :  
    System.Collections.Generic.ICollection< TKey >,  
    System.Collections.Generic.IEnumerable< TKey >,  
    System.Collections.Generic.IReadOnlyCollection< TKey >,  
    System.Collections.ICollection
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [SortedDictionary< TKey, TValue >.KeyCollection](#)

Implements [ICollection< TKey >](#) , [IEnumerable< T >](#) , [IEnumerable< TKey >](#) ,
[IReadOnlyCollection< TKey >](#) , [ICollection](#) , [IEnumerable](#)

Remarks

The [SortedDictionary< TKey, TValue >.Keys](#) property returns an instance of this type, containing all the keys in that [SortedDictionary< TKey, TValue >](#). The order of the keys in the [SortedDictionary< TKey, TValue >.KeyCollection](#) is the same as the order of elements in the [SortedDictionary< TKey, TValue >](#), the same as the order of the associated values in the [SortedDictionary< TKey, TValue >.ValueCollection](#) returned by the [SortedDictionary< TKey, TValue >.Values](#) property.

The `SortedDictionary< TKey, TValue >.KeyCollection` is not a static copy; instead, the `SortedDictionary< TKey, TValue >.KeyCollection` refers back to the keys in the original `SortedDictionary< TKey, TValue >`. Therefore, changes to the `SortedDictionary< TKey, TValue >` continue to be reflected in the `SortedDictionary< TKey, TValue >.KeyCollection`.

Constructors

[+] Expand table

<code>SortedDictionary< TKey, TValue >.KeyCollection(SortedDictionary< TKey, TValue >)</code>	Initializes a new instance of the <code>SortedDictionary< TKey, TValue >.KeyCollection</code> class that reflects the keys in the specified <code>SortedDictionary< TKey, TValue ></code> .
---	---

Properties

[+] Expand table

<code>Count</code>	Gets the number of elements contained in the <code>SortedDictionary< TKey, TValue >.KeyCollection</code> .
--------------------	--

Methods

[+] Expand table

<code>CopyTo(TKey[], Int32)</code>	Copies the <code>SortedDictionary< TKey, TValue >.KeyCollection</code> elements to an existing one-dimensional array, starting at the specified array index.
<code>Equals(Object)</code>	Determines whether the specified object is equal to the current object. (Inherited from <code>Object</code>)
<code>GetEnumerator()</code>	Returns an enumerator that iterates through the <code>SortedDictionary< TKey, TValue >.KeyCollection</code> .
<code>GetHashCode()</code>	Serves as the default hash function. (Inherited from <code>Object</code>)
<code>GetType()</code>	Gets the <code>Type</code> of the current instance. (Inherited from <code>Object</code>)
<code>MemberwiseClone()</code>	Creates a shallow copy of the current <code>Object</code> . (Inherited from <code>Object</code>)
<code>ToString()</code>	Returns a string that represents the current object.

[+] Expand table

Explicit Interface Implementations

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an array, starting at a particular array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
ICollection<TKey>.Add(TKey)	Adds an item to the ICollection<T> . This implementation always throws a NotSupportedException .
ICollection<TKey>.Clear()	Removes all items from the ICollection<T> . This implementation always throws a NotSupportedException .
ICollection<TKey>.Contains(TKey)	Determines whether the ICollection<T> contains the specified value.
ICollection<TKey>.IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.
ICollection<TKey>.Remove(TKey)	Removes the first occurrence of a specific object from the ICollection<T> . This implementation always throws a NotSupportedException .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through the collection.
IEnumerable<TKey>.Get Enumerator()	Returns an enumerator that iterates through the collection.

Extension Methods

[+] Expand table

TolImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.

<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.

<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEqualityComparer<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEqualityComparer<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.

<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable <code>Int32</code> values that are obtained by invoking a

	transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.

DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.

<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.

<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.

<code>Intersect<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEqualityComparer<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEqualityComparer<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEqualityComparer<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEqualityComparer<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IQueryable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IQueryable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IQueryable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.

<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Decimal</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Double</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int64</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Decimal</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Double</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum nullable Int64 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the maximum Single value.
Max<TSource>(IEnumerable<TSource>)	Returns the maximum value in a generic sequence.
Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function.
Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.

<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> ,

<code>Func<TSource, TCollection, TResult></code>	flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified <code>IEqualityComparer<T></code> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
<code>Single<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
<code>Single<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than

	one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.

Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.
Take<TSource>(IEnumerable<TSource>, Range)	Returns a specified range of contiguous elements from a

	sequence.
TakeLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns elements from a sequence as long as a specified condition is true.
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a <code>IEnumerable<T></code> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
ToList<TSource>(IEnumerable<TSource>)	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .

ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an IEnumerable to an IQueryable .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of every node in the source collection.
<code>DescendantNodes<T>(IEnumerable<T>)</code>	Returns a collection of the descendant nodes of every document and element in the source collection.
<code>Descendants<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
<code>Descendants<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the descendant

	elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [SortedDictionary< TKey, TValue >.KeyCollection](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the

collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

SortedDictionary<TKey,TValue>.KeyCollection Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [SortedDictionary<TKey,TValue>.KeyCollection](#) class that reflects the keys in the specified [SortedDictionary<TKey,TValue>](#).

C#

```
public KeyCollection(System.Collections.Generic.SortedDictionary<TKey, TValue>
dictionary);
```

Parameters

dictionary [SortedDictionary<TKey,TValue>](#)

The [SortedDictionary<TKey,TValue>](#) whose keys are reflected in the new [SortedDictionary<TKey,TValue>.KeyCollection](#).

Exceptions

[ArgumentNullException](#)

dictionary is `null`.

Remarks

The [SortedDictionary<TKey,TValue>.KeyCollection](#) is not a static copy; instead, the [SortedDictionary<TKey,TValue>.KeyCollection](#) refers back to the keys in the original [SortedDictionary<TKey,TValue>](#). Therefore, changes to the [SortedDictionary<TKey,TValue>](#) continue to be reflected in the [SortedDictionary<TKey,TValue>.KeyCollection](#).

This constructor is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.KeyCollection.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements contained in the [SortedDictionary< TKey, TValue >.KeyCollection](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [SortedDictionary< TKey, TValue >.KeyCollection](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Remarks

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.KeyCollection.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [SortedDictionary< TKey, TValue >.KeyCollection](#) elements to an existing one-dimensional array, starting at the specified array index.

C#

```
public void CopyTo(TKey[] array, int index);
```

Parameters

array `TKey[]`

The one-dimensional array that is the destination of the elements copied from the [SortedDictionary< TKey, TValue >.KeyCollection](#). The array must have zero-based indexing.

index `Int32`

The zero-based index in `array` at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#)

`index` is less than 0.

[ArgumentException](#)

The number of elements in the source [SortedDictionary< TKey, TValue >.KeyCollection](#) is greater than the available space from `index` to the end of the destination `array`.

Remarks

The elements are copied to the array in the same order in which the enumerator iterates through the `SortedDictionary< TKey, TValue >.KeyCollection`.

This method is an $O(n)$ operation, where n is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Array](#)

SortedDictionary<TKey,TValue>.KeyCollection.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [SortedDictionary<TKey,TValue>.KeyCollection](#).

C#

```
public  
System.Collections.Generic.SortedDictionary<TKey,TValue>.KeyCollection.Enumerator  
GetEnumerator();
```

Returns

[SortedDictionary<TKey,TValue>.KeyCollection.Enumerator](#)

A [SortedDictionary<TKey,TValue>.KeyCollection.Enumerator](#) structure for the [SortedDictionary<TKey,TValue>.KeyCollection](#).

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, `Current` is undefined. You must call the `MoveNext` method to advance the enumerator to the first element of the collection before reading the value of `Current`.

The `Current` property returns the same object until `MoveNext` is called. `MoveNext` sets `Current` to the next element.

If `MoveNext` passes the end of the collection, the enumerator is positioned after the last element in the collection and `MoveNext` returns `false`. When the enumerator is at this

position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [IEnumerator<T>](#)

SortedDictionary<TKey,TValue>.Key Collection.ICollection<TKey>.Add Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to the [ICollection<T>](#). This implementation always throws a [NotSupportedException](#).

C#

```
void ICollection<TKey>.Add(TKey item);
```

Parameters

item TKey

The object to add to the [ICollection<T>](#).

Implements

[Add\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown; the collection is read-only.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

SortedDictionary<TKey, TValue>.Key Collection.ICollection<TKey>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all items from the [ICollection<T>](#). This implementation always throws a [NotSupportedException](#).

C#

```
void ICollection<TKey>.Clear();
```

Implements

[Clear\(\)](#)

Exceptions

[NotSupportedException](#)

Always thrown; the collection is read-only.

Remarks

The [Count](#) property is set to 0, and references to other objects from elements of the collection are also released.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

SortedDictionary<TKey,TValue>.Key Collection.ICollection<TKey>.Contains Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [ICollection<T>](#) contains the specified value.

C#

```
bool ICollection<TKey>.Contains(TKey item);
```

Parameters

item TKey

The object to locate in the [ICollection<T>](#).

Returns

Boolean

`true` if item is found in the [ICollection<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas [SortedDictionary<TKey,TValue>](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.KeyCollection.ICollection< TKey >.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool System.Collections.Generic.ICollection<TKey>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >.KeyCollection](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey, TValue>.KeyCollection.ICollection<TKey>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a specific object from the [ICollection<T>](#). This implementation always throws a [NotSupportedException](#).

C#

```
bool ICollection<TKey>.Remove(TKey item);
```

Parameters

item TKey

The object to remove from the [ICollection<T>](#).

Returns

Boolean

`true` if item is successfully removed from the [ICollection<T>](#); otherwise, `false`. This method also returns `false` if item is not found in the [ICollection<T>](#).

Implements

[Remove\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown; the collection is read-only.

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas [SortedDictionary< TKey, TValue >](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

SortedDictionary<TKey, TValue>.KeyCollection.IEnumerable<TKey>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.Generic.IEnumerator<TKey> IEnumerable<TKey>.GetEnumerator();
```

Returns

[IEnumerator<TKey>](#)

An [IEnumerator<T>](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns false. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return false. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

SortedDictionary< TKey, TValue >.Key Collection.ICollection.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an array, starting at a particular array index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional array that is the destination of the elements copied from the [ICollection](#).
The array must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than 0.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the nongeneric implementations of [ICollection.CopyTo](#) throw an [InvalidOperationException](#), whereas the generic implementations throw an [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)
- [Array](#)

SortedDictionary< TKey, TValue >.KeyCollection.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >.KeyCollection](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

SortedDictionary< TKey, TValue >.KeyCollection.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [SortedDictionary< TKey, TValue >.KeyCollection](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;
lock (ic.SyncRoot)
{
    // Access the collection.
}
```

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

SortedDictionary< TKey, TValue >.Key Collection.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.IEnumerable IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until either [MoveNext](#) or [Reset](#) is called.

[MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator](#)

SortedDictionary< TKey, TValue >.ValueCollection.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [SortedDictionary< TKey, TValue >.ValueCollection](#).

C#

```
public struct SortedDictionary< TKey, TValue >.ValueCollection.Enumerator :  
    System.Collections.Generic.IEnumerator< TValue >
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [ValueType](#) →
[SortedDictionary< TKey, TValue >.ValueCollection.Enumerator](#)

Implements [IEnumerator< TValue >](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator. This type implements the [IEnumerator< T >](#) interface.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Properties

 [Expand table](#)

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

 [Expand table](#)

Dispose()	Releases all resources used by the SortedDictionary<TKey,TValue>.ValueCollection.Enumerator .
MoveNext()	Advances the enumerator to the next element of the SortedDictionary<TKey,TValue>.ValueCollection .

Explicit Interface Implementations

IEnumerator.Current	Gets the element at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.Enumerator](#)
- [SortedDictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

SortedDictionary< TKey, TValue >.ValueCollection.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public TValue Current { get; }
```

Property Value

TValue

The element in the [SortedDictionary< TKey, TValue >.ValueCollection](#) at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.ValueCollection.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [SortedDictionary< TKey, TValue >.ValueCollection.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue>.ValueCollection.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [SortedDictionary<TKey,TValue>.ValueCollection](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to the [MoveNext](#) method advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

SortedDictionary< TKey, TValue >.ValueCollection.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.

- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IEnumerator.Current](#) does not move the position of the enumerator, and consecutive calls to [IEnumerator.Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.ValueCollection.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumerator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After calling the [IEnumerator.Reset](#) method, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)
- [MoveNext\(\)](#)
- [Reset\(\)](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.ValueCollection Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents the collection of values in a [SortedDictionary< TKey, TValue >](#). This class cannot be inherited.

C#

```
public sealed class SortedDictionary< TKey, TValue >.ValueCollection :  
    System.Collections.Generic.ICollection< TValue >,  
    System.Collections.Generic.IEnumerable< TValue >,  
    System.Collections.Generic.IReadOnlyCollection< TValue >,  
    System.Collections.ICollection
```

Type Parameters

TKey

TValue

Inheritance [Object](#) → [SortedDictionary< TKey, TValue >.ValueCollection](#)

Implements [ICollection< TValue >](#) , [IEnumerable< T >](#) , [IEnumerable< TValue >](#) ,
[IReadOnlyCollection< TValue >](#) , [ICollection](#) , [IEnumerable](#)

Remarks

The [SortedDictionary< TKey, TValue >.Values](#) property returns an instance of this type, containing all the values in that [SortedDictionary< TKey, TValue >](#). The order of the values in the [SortedDictionary< TKey, TValue >.ValueCollection](#) is the same as the order of the elements in the [SortedDictionary< TKey, TValue >](#), and the same as the order of the associated keys in the [SortedDictionary< TKey, TValue >.KeyCollection](#) returned by the [SortedDictionary< TKey, TValue >.Keys](#) property.

The `SortedDictionary< TKey, TValue >.ValueCollection` is not a static copy; instead, the `SortedDictionary< TKey, TValue >.ValueCollection` refers back to the values in the original `SortedDictionary< TKey, TValue >`. Therefore, changes to the `SortedDictionary< TKey, TValue >` continue to be reflected in the `SortedDictionary< TKey, TValue >.ValueCollection`.

Constructors

[] Expand table

<code>SortedDictionary< TKey, TValue >.ValueCollection(SortedDictionary< TKey, TValue >)</code>	Initializes a new instance of the <code>SortedDictionary< TKey, TValue >.ValueCollection</code> class that reflects the values in the specified <code>SortedDictionary< TKey, TValue ></code> .
---	---

Properties

[] Expand table

<code>Count</code>	Gets the number of elements contained in the <code>SortedDictionary< TKey, TValue >.ValueCollection</code> .
--------------------	--

Methods

[] Expand table

<code>CopyTo(TValue[], Int32)</code>	Copies the <code>SortedDictionary< TKey, TValue >.ValueCollection</code> elements to an existing one-dimensional array, starting at the specified array index.
<code>Equals(Object)</code>	Determines whether the specified object is equal to the current object. (Inherited from <code>Object</code>)
<code>GetEnumerator()</code>	Returns an enumerator that iterates through the <code>SortedDictionary< TKey, TValue >.ValueCollection</code> .
<code>GetHashCode()</code>	Serves as the default hash function. (Inherited from <code>Object</code>)
<code>GetType()</code>	Gets the <code>Type</code> of the current instance. (Inherited from <code>Object</code>)
<code>MemberwiseClone()</code>	Creates a shallow copy of the current <code>Object</code> . (Inherited from <code>Object</code>)
<code>ToString()</code>	Returns a string that represents the current object.

[+] Expand table

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an array, starting at a particular array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
ICollection< TValue >.Add(TValue)	Adds an item to the ICollection< T > . This implementation always throws a NotSupportedException .
ICollection< TValue >.Clear()	Removes all items from the ICollection< T > . This implementation always throws a NotSupportedException .
ICollection< TValue >.Contains(TValue)	Determines whether the ICollection< T > contains a specified value.
ICollection< TValue >.IsReadOnly	Gets a value indicating whether the ICollection< T > is read-only.
ICollection< TValue >.Remove(TValue)	Removes the first occurrence of a specific object from the ICollection< T > . This implementation always throws a NotSupportedException .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through the collection.
IEnumerable< TValue >.Get Enumerator()	Returns an enumerator that iterates through the collection.

Extension Methods

[+] Expand table

TolImmutableArray< TSource >(IEnumerable< TSource >)	Creates an immutable array from the specified collection.
TolImmutableDictionary< TSource, TKey >(IEnumerable< TSource >, Func< TSource, TKey >, IEqualityComparer< TKey >)	Constructs an immutable dictionary based on some transformation of a sequence.

<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.

<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEqualityComparer<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEqualityComparer<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEqualityComparer<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
<code>All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether all elements of a sequence satisfy a condition.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.

<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Append<TSource>(IEnumerable<TSource>, TSource)</code>	Appends a value to the end of the sequence.
<code>AsEnumerable<TSource>(IEnumerable<TSource>)</code>	Returns the input typed as <code>IEnumerable<T></code> .
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable <code>Int32</code> values that are obtained by invoking a

	transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.

DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.

<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.

<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.

<code>Intersect<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEqualityComparer<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEqualityComparer<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEqualityComparer<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEqualityComparer<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IQueryable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IQueryable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IQueryable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IQueryable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.

<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Decimal</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Double</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum <code>Int64</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Decimal</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Double</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable <code>Int32</code> value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum nullable Int64 value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Invokes a transform function on each element of a sequence and returns the maximum Single value.
Max<TSource>(IEnumerable<TSource>)	Returns the maximum value in a generic sequence.
Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns the maximum value in a generic sequence according to a specified key selector function.
Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Invokes a transform function on each element of a sequence and returns the minimum Double value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.
<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.

<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> ,

<code>Func<TSource, TCollection, TResult>()</code>	flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing their elements by using a specified <code>IEqualityComparer<T></code> .
<code>SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
<code>Single<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
<code>Single<TSource>(IEnumerable<TSource>)</code>	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
<code>SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than

	one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.

Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.
Take<TSource>(IEnumerable<TSource>, Range)	Returns a specified range of contiguous elements from a

	sequence.
TakeLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns elements from a sequence as long as a specified condition is true.
TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a <code>IEnumerable<T></code> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
ToList<TSource>(IEnumerable<TSource>)	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .

ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an IEnumerable to an IQueryable .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of every node in the source collection.
<code>DescendantNodes<T>(IEnumerable<T>)</code>	Returns a collection of the descendant nodes of every document and element in the source collection.
<code>Descendants<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
<code>Descendants<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the descendant

	elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [SortedDictionary<TKey,TValue>.ValueCollection](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you

can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

SortedDictionary< TKey, TValue >.ValueCollection Constructor

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [SortedDictionary< TKey, TValue >.ValueCollection](#) class that reflects the values in the specified [SortedDictionary< TKey, TValue >](#).

C#

```
public ValueCollection(System.Collections.Generic.SortedDictionary< TKey, TValue >
dictionary);
```

Parameters

dictionary [SortedDictionary< TKey, TValue >](#)

The [SortedDictionary< TKey, TValue >](#) whose values are reflected in the new [SortedDictionary< TKey, TValue >.ValueCollection](#).

Exceptions

[ArgumentNullException](#)

dictionary is `null`.

Remarks

The [SortedDictionary< TKey, TValue >.ValueCollection](#) is not a static copy; instead, the [SortedDictionary< TKey, TValue >.ValueCollection](#) refers back to the values in the original [SortedDictionary< TKey, TValue >](#). Therefore, changes to the [SortedDictionary< TKey, TValue >](#) continue to be reflected in the [SortedDictionary< TKey, TValue >.ValueCollection](#).

This constructor is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.ValueCollection.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements contained in the [SortedDictionary< TKey, TValue >.ValueCollection](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [SortedDictionary< TKey, TValue >.ValueCollection](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Remarks

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.ValueCollection.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [SortedDictionary< TKey, TValue >.ValueCollection](#) elements to an existing one-dimensional array, starting at the specified array index.

C#

```
public void CopyTo(TValue[] array, int index);
```

Parameters

array TValue[]

The one-dimensional array that is the destination of the elements copied from the [SortedDictionary< TKey, TValue >.ValueCollection](#). The array must have zero-based indexing.

index Int32

The zero-based index in `array` at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#)

`index` is less than 0.

[ArgumentException](#)

The number of elements in the source [SortedDictionary< TKey, TValue >.ValueCollection](#) is greater than the available space from `index` to the end of the destination `array`.

Remarks

The elements are copied to the array in the same order in which the enumerator iterates through the `SortedDictionary< TKey, TValue >.ValueCollection`.

This method is an $O(n)$ operation, where n is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Array](#)

SortedDictionary<TKey,TValue>.ValueCollection.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [SortedDictionary<TKey,TValue>.ValueCollection](#).

C#

```
public  
System.Collections.Generic.SortedDictionary<TKey,TValue>.ValueCollection.Enumerator  
GetEnumerator();
```

Returns

[SortedDictionary<TKey,TValue>.ValueCollection.Enumerator](#)

A [SortedDictionary<TKey,TValue>.ValueCollection.Enumerator](#) structure for the [SortedDictionary<TKey,TValue>.ValueCollection](#).

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. You must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this

position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [IEnumerator<T>](#)

SortedDictionary<TKey, TValue>.ValueCollection.ICollection<TValue>.Add Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to the [ICollection<T>](#). This implementation always throws a [NotSupportedException](#).

C#

```
void ICollection<TValue>.Add(TValue item);
```

Parameters

item TValue

The object to add to the [ICollection<T>](#).

Implements

[Add\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown; the collection is read-only.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [IsReadOnly](#)

SortedDictionary<TKey, TValue>.ValueCollection.ICollection<TValue>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all items from the [ICollection<T>](#). This implementation always throws a [NotSupportedException](#).

C#

```
void ICollection<TValue>.Clear();
```

Implements

[Clear\(\)](#)

Exceptions

[NotSupportedException](#)

Always thrown; the collection is read-only.

Remarks

The [Count](#) property is set to 0, and references to other objects from elements of the collection are also released.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [IsReadOnly](#)

SortedDictionary< TKey, TValue >.ValueCollection.ICollection< TValue >.Contains Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [ICollection<T>](#) contains a specified value.

C#

```
bool ICollection<TValue>.Contains(TValue item);
```

Parameters

item TValue

The object to locate in the [ICollection<T>](#).

Returns

Boolean

`true` if `item` is found in the [ICollection<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas [SortedDictionary< TKey, TValue >](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.ValueCollection.ICollection< TValue >.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool System.Collections.Generic.ICollection<TValue>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >.ValueCollection](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey, TValue>.ValueCollection.ICollection<TValue>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a specific object from the [ICollection<T>](#). This implementation always throws a [NotSupportedException](#).

C#

```
bool ICollection<TValue>.Remove(TValue item);
```

Parameters

item TValue

The object to remove from the [ICollection<T>](#).

Returns

Boolean

`true` if `item` is successfully removed from the [ICollection<T>](#); otherwise, `false`. This method also returns `false` if `item` is not found in the [ICollection<T>](#).

Implements

[Remove\(T\)](#)

Exceptions

[NotSupportedException](#)

Always thrown; the collection is read-only.

Remarks

Implementations can vary in how they determine equality of objects; for example, [List<T>](#) uses [Default](#), whereas [SortedDictionary< TKey, TValue >](#) allows the user to specify the [IComparer<T>](#) implementation to use for comparing keys.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)

SortedDictionary< TKey, TValue >.ValueCollection.IEnumerable< TValue >.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.Generic.IEnumerator<TValue>
IEnumerable<TValue>.GetEnumerator();
```

Returns

[IEnumerator< TValue >](#)

An [IEnumerator< T >](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

SortedDictionary< TKey, TValue >.ValueCollection.ICollection.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an array, starting at a particular array index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional array that is the destination of the elements copied from the [ICollection](#).
The array must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than 0.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the nongeneric implementations of [ICollection.CopyTo](#) throw an [InvalidOperationException](#), whereas the generic implementations throw an [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)
- [Array](#)

SortedDictionary< TKey, TValue >.ValueCollection.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >.ValueCollection](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

SortedDictionary< TKey, TValue >.ValueCollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [SortedDictionary< TKey, TValue >.ValueCollection](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;
lock (ic.SyncRoot)
{
    // Access the collection.
}
```

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

SortedDictionary< TKey, TValue >.ValueCollection.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.IEnumerable IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same object until either [MoveNext](#) or [Reset](#) is called.

[MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator](#)

SortedDictionary<TKey,TValue> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a collection of key/value pairs that are sorted on the key.

C#

```
public class SortedDictionary<TKey, TValue> :  
    System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>, System.Collections.Generic.IDictionary<TKey, TValue>,  
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>>,  
    System.Collections.Generic.IReadOnlyCollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>, System.Collections.Generic.IReadOnlyDictionary<TKey, TValue>,  
    System.Collections.IDictionary
```

Type Parameters

TKey

The type of the keys in the dictionary.

TValue

The type of the values in the dictionary.

Inheritance [Object](#) → [SortedDictionary<TKey,TValue>](#)

Implements [ICollection<KeyValuePair<TKey,TValue>>](#) , [IDictionary<TKey,TValue>](#) ,
[IEnumerable<KeyValuePair<TKey,TValue>>](#) , [IEnumerable<T>](#) ,
[IReadOnlyCollection<KeyValuePair<TKey,TValue>>](#) ,
[IReadOnlyDictionary<TKey,TValue>](#) , [ICollection](#) , [IDictionary](#) , [IEnumerable](#)

Examples

The following code example creates an empty [SortedDictionary<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.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string
        // keys.
        SortedDictionary<string, string> openWith =
            new SortedDictionary<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.
SortedDictionary<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.
SortedDictionary<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 = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = doc, Value = winword.exe
Key = ht, Value = hypertrm.exe
Key = rtf, Value = winword.exe
Key = txt, Value = notepad.exe

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

Key = bmp

```

```
Key = dib
Key = doc
Key = ht
Key = rtf
Key = txt

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

Remarks

The [SortedDictionary<TKey,TValue>](#) generic class is a binary search tree with $O(\log n)$ retrieval, where n is the number of elements in the dictionary. In this respect, it is similar to the [SortedList<TKey,TValue>](#) generic class. The two classes have similar object models, and both have $O(\log n)$ retrieval. Where the two classes differ is in memory use and speed of insertion and removal:

- [SortedList<TKey,TValue>](#) uses less memory than [SortedDictionary<TKey,TValue>](#).
- [SortedDictionary<TKey,TValue>](#) has faster insertion and removal operations for unsorted data: $O(\log n)$ as opposed to $O(n)$ for [SortedList<TKey,TValue>](#).
- If the list is populated all at once from sorted data, [SortedList<TKey,TValue>](#) is faster than [SortedDictionary<TKey,TValue>](#).

Each key/value pair can be retrieved as a [KeyValuePair<TKey,TValue>](#) structure, or as a [DictionaryEntry](#) through the nongeneric [IDictionary](#) interface.

Keys must be immutable as long as they are used as keys in the [SortedDictionary<TKey,TValue>](#). Every key in a [SortedDictionary<TKey,TValue>](#) must be unique. A key cannot be `null`, but a value can be, if the value type `TValue` is a reference type.

[SortedDictionary<TKey,TValue>](#) requires a comparer implementation to perform key comparisons. You can specify an implementation of the [IComparer<T>](#) generic interface by using a constructor that accepts a `comparer` parameter; if you do not specify an implementation, the default generic comparer [Comparer<T>.Default](#) is used. If type `TKey` implements the [System.IComparable<T>](#) generic interface, the default comparer uses that implementation.

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 [SortedDictionary<TKey,TValue>](#) 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 [KeyValuePair<TKey,TValue>](#). The following code shows the syntax.

C#

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

The `foreach` statement is a wrapper around the enumerator, which allows only reading from the collection, not writing to it.

Constructors

[+] [Expand table](#)

SortedDictionary<TKey,TValue>()	Initializes a new instance of the SortedDictionary<TKey,TValue> class that is empty and uses the default IComparer<T> implementation for the key type.
SortedDictionary<TKey,TValue>(IComparer<TKey>)	Initializes a new instance of the SortedDictionary<TKey,TValue> class that is empty and uses the specified IComparer<T> implementation to compare keys.
SortedDictionary<TKey,TValue>(IDictionary<TKey,TValue>, IComparer<TKey>)	Initializes a new instance of the SortedDictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the specified IComparer<T> implementation to compare keys.
SortedDictionary<TKey,TValue>(IDictionary<TKey,TValue>)	Initializes a new instance of the SortedDictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the default IComparer<T> implementation for the key type.

Properties

[+] [Expand table](#)

Comparer	Gets the IComparer<T> used to order the elements of the SortedDictionary<TKey,TValue> .
Count	Gets the number of key/value pairs contained in the SortedDictionary<TKey,TValue> .

Item(TKey)	Gets or sets the value associated with the specified key.
Keys	Gets a collection containing the keys in the SortedDictionary<TKey,TValue> .
Values	Gets a collection containing the values in the SortedDictionary<TKey,TValue> .

Methods

 [Expand table](#)

Add(TKey, TValue)	Adds an element with the specified key and value into the SortedDictionary<TKey,TValue> .
Clear()	Removes all elements from the SortedDictionary<TKey,TValue> .
ContainsKey(TKey)	Determines whether the SortedDictionary<TKey,TValue> contains an element with the specified key.
ContainsValue(TValue)	Determines whether the SortedDictionary<TKey,TValue> contains an element with the specified value.
CopyTo(KeyValuePair<TKey,TValue>[], Int32)	Copies the elements of the SortedDictionary<TKey,TValue> to the specified array of KeyValuePair<TKey,TValue> structures, starting at the specified index.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the SortedDictionary<TKey,TValue> .
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)
Remove(TKey)	Removes the element with the specified key from the SortedDictionary<TKey,TValue> .
ToString()	Returns a string that represents the current object. (Inherited from Object)
TryGetValue(TKey, TValue)	Gets the value associated with the specified key.

Explicit Interface Implementations

 Expand table

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection<T> to an array, starting at the specified array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
ICollection<KeyValuePair< TKey, TValue >>.Add(KeyValuePair< TKey, TValue >)	Adds an item to the ICollection<T> .
ICollection<KeyValuePair< TKey, TValue >>.Contains(KeyValuePair< TKey, TValue >)	Determines whether the ICollection<T> contains a specific key and value.
ICollection<KeyValuePair< TKey, TValue >>.IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.
ICollection<KeyValuePair< TKey, TValue >>.Remove(KeyValuePair< TKey, TValue >)	Removes the first occurrence of the specified element from the ICollection<T> .
IDictionary.Add(Object, Object)	Adds an element with the provided key and value to the IDictionary .
IDictionary.Contains(Object)	Determines whether the IDictionary contains an element with the specified key.
IDictionary.GetEnumerator()	Returns an IDictionaryEnumerator for the IDictionary .
IDictionary.IsFixedSize	Gets a value indicating whether the IDictionary has a fixed size.
IDictionary.IsReadOnly	Gets a value indicating whether the IDictionary is read-only.
IDictionary.Item[Object]	Gets or sets the element with the specified key.
IDictionary.Keys	Gets an ICollection containing the keys of the IDictionary .
IDictionary.Remove(Object)	Removes the element with the specified key from the IDictionary .
IDictionary.Values	Gets an ICollection containing the values in the IDictionary .

IDictionary<TKey,TValue>.Keys	Gets an ICollection<T> containing the keys of the IDictionary<TKey,TValue> .
IDictionary<TKey,TValue>.Values	Gets an ICollection<T> containing the values in the IDictionary<TKey,TValue> .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through the collection.
IEnumerable<KeyValuePair<TKey,TValue>>.GetEnumerator()	Returns an enumerator that iterates through a collection.
 IReadOnlyDictionary<TKey,TValue>.Keys	Gets a collection containing the keys in the SortedDictionary<TKey,TValue> .
 IReadOnlyDictionary<TKey,TValue>.Values	Gets a collection containing the values in the SortedDictionary<TKey,TValue> .

Extension Methods

[] [Expand table](#)

GetValueOrDefault<TKey,TValue>(IReadOnlyDictionary<TKey,TValue>, TKey, TValue)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
GetValueOrDefault<TKey,TValue>(IReadOnlyDictionary<TKey,TValue>, TKey)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
Remove<TKey,TValue>(IDictionary<TKey,TValue>, TKey, TValue)	Tries to remove the value with the specified <code>key</code> from the <code>dictionary</code> .
TryAdd<TKey,TValue>(IDictionary<TKey,TValue>, TKey, TValue)	Tries to add the specified <code>key</code> and <code>value</code> to the <code>dictionary</code> .
ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.

<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its

	contents, and uses the specified comparer.
ToImmutableSortedSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable sorted set of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillErrorEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T> object where the generic parameter <code>T</code> is DataRow .
Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)	Applies an accumulator function over a sequence.
Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether all elements of a sequence satisfy a condition.
Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether any element of a sequence satisfies a condition.
Any<TSource>(IEnumerable<TSource>)	Determines whether a sequence contains any elements.
Append<TSource>(IEnumerable<TSource>, TSource)	Appends a value to the end of the sequence.
AsEnumerable<TSource>(IEnumerable<TSource>)	Returns the input typed as

`IEnumerable<T>`.

<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Computes the average of a sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Computes the average of a sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Computes the average of a sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the average of a sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the average of a sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the average of a sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the average of a sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the average of a sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.

Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified

	<code>IEqualityComparer<T></code> to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
<code>DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns distinct elements from a sequence according to a specified key selector function.
<code>ElementAt<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence.
<code>ElementAt<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)</code>	Returns the element at a specified index in a sequence or a default value if the index is out of range.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified

	condition.
First<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence.
FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
FirstOrDefault<TSource>(IEnumerable<TSource>)	Returns the first element of a sequence, or a default value if the sequence contains no elements.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Groups the elements of a sequence according to a specified key selector function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a

	<p>result value from each group and its key. The keys are compared by using a specified comparer.</p>
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.</p>
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	<p>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	<p>Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.</p>
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)</code>	<p>Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	<p>Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.</p>
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	<p>Produces the set intersection of two sequences by using the default equality comparer to compare values.</p>

<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOOuter,TInner,TKey,TResult>(IEnumerable<TOOuter>, IEnumerable<TInner>, Func<TOOuter,TKey>, Func<TInner,TKey>, Func<TOOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a

	sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.

<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.

<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEqualityComparer<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing

	their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then

	returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is

	true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.

ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.

InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`Shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [SortedDictionary<TKey,TValue>](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

See also

- [SortedList](#)
- [Dictionary<TKey,TValue>](#)

SortedDictionary<TKey,TValue>

Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [SortedDictionary<TKey,TValue>](#) class.

Overloads

 [Expand table](#)

SortedDictionary<TKey,TValue>()	Initializes a new instance of the SortedDictionary<TKey,TValue> class that is empty and uses the default IComparer<T> implementation for the key type.
SortedDictionary<TKey,TValue>(IComparer<TKey>)	Initializes a new instance of the SortedDictionary<TKey,TValue> class that is empty and uses the specified IComparer<T> implementation to compare keys.
SortedDictionary<TKey,TValue>(IDictionary<TKey,TValue>)	Initializes a new instance of the SortedDictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the default IComparer<T> implementation for the key type.
SortedDictionary<TKey,TValue>(IDictionary<TKey,TValue>, IComparer<TKey>)	Initializes a new instance of the SortedDictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the specified IComparer<T> implementation to compare keys.

SortedDictionary<TKey,TValue>()

Initializes a new instance of the [SortedDictionary<TKey,TValue>](#) class that is empty and uses the default [IComparer<T>](#) implementation for the key type.

C#

```
public SortedDictionary();
```

Examples

The following code example creates an empty [SortedDictionary<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.

This code example is part of a larger example provided for the [SortedDictionary<TKey,TValue>](#) class.

C#

```
// Create a new sorted dictionary of strings, with string
// keys.
SortedDictionary<string, string> openWith =
    new SortedDictionary<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.");
}
```

Remarks

Every key in a [SortedDictionary<TKey,TValue>](#) must be unique according to the default comparer.

[SortedDictionary<TKey,TValue>](#) requires a comparer implementation to perform key comparisons. This constructor uses the default generic equality comparer [Comparer<T>.Default](#). If type [TKey](#) implements the [System.IComparable<T>](#) generic interface, the default comparer uses that implementation. Alternatively, you can specify an implementation of the [IComparer<T>](#) generic interface by using a constructor that accepts a [comparer](#) parameter.

This constructor is an O(1) operation.

See also

- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue > (IComparer< TKey >)

Initializes a new instance of the [SortedDictionary< TKey, TValue >](#) class that is empty and uses the specified [IComparer< T >](#) implementation to compare keys.

C#

```
public SortedDictionary(System.Collections.Generic.IComparer< TKey >? comparer);
```

Parameters

comparer [IComparer< TKey >](#)

The [IComparer< T >](#) implementation to use when comparing keys, or `null` to use the default [Comparer< T >](#) for the type of the key.

Examples

The following code example creates a `SortedDictionary< TKey, TValue >` with a case-insensitive comparer for the current culture. The example adds four elements, some with lower-case keys and some with upper-case keys. The example then attempts to add an element with a key that differs from an existing key only by case, catches the resulting exception, and displays an error message. Finally, the example displays the elements in case-insensitive sort order.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new SortedDictionary of strings, with string keys
        // and a case-insensitive comparer for the current culture.
        SortedDictionary<string, string> openWith =
            new SortedDictionary<string, string>(
                StringComparer.CurrentCultureIgnoreCase);

        // Add some elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("DIB", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Try to add a fifth element with a key that is the same
        // except for case; this would be allowed with the default
        // comparer.
        try
        {
            openWith.Add("BMP", "paint.exe");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("\nBMP is already in the dictionary.");
        }

        // List the contents of the sorted dictionary.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
                kvp.Value);
        }
    }

    /* This code example produces the following output:
    
```

```
BMP is already in the dictionary.
```

```
Key = bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/
```

Remarks

Every key in a [SortedDictionary<TKey,TValue>](#) must be unique according to the specified comparer.

[SortedDictionary<TKey,TValue>](#) requires a comparer implementation to perform key comparisons. If `comparer` is `null`, this constructor uses the default generic equality comparer, [Comparer<T>.Default](#). If type `TKey` implements the [System.IComparable<T>](#) generic interface, the default comparer uses that implementation.

This constructor is an O(1) operation.

See also

- [IComparer<T>](#)
- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue> (IDictionary<TKey,TValue>)

Initializes a new instance of the [SortedDictionary<TKey,TValue>](#) class that contains elements copied from the specified [IDictionary<TKey,TValue>](#) and uses the default [IComparer<T>](#) implementation for the key type.

C#

```
public SortedDictionary(System.Collections.Generic.IDictionary<TKey, TValue>
dictionary);
```

Parameters

dictionary [IDictionary<TKey,TValue>](#)

The [IDictionary<TKey,TValue>](#) whose elements are copied to the new [SortedDictionary<TKey,TValue>](#).

Exceptions

[ArgumentNullException](#)

dictionary is `null`.

[ArgumentException](#)

dictionary contains one or more duplicate keys.

Examples

The following code example shows how to use [SortedDictionary<TKey,TValue>](#) to create a sorted copy of the information in a [Dictionary<TKey,TValue>](#), by passing the [Dictionary<TKey,TValue>](#) to the [SortedDictionary<TKey,TValue>\(IComparer<TKey>\)](#) constructor.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new Dictionary of strings, with string keys.
        //
        Dictionary<string, string> openWith =
            new Dictionary<string, string>();

        // Add some elements to the dictionary.
```

```

        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Create a SortedDictionary of strings with string keys,
        // and initialize it with the contents of the Dictionary.
        SortedDictionary<string, string> copy =
            new SortedDictionary<string, string>(openWith);

        // List the contents of the copy.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in copy )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }

/* This code example produces the following output:

Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/

```

Remarks

Every key in a [SortedDictionary<TKey,TValue>](#) must be unique according to the default comparer; therefore, every key in the source `dictionary` must also be unique according to the default comparer.

`SortedDictionary<TKey,TValue>` requires a comparer implementation to perform key comparisons. This constructor uses the default generic equality comparer, [Comparer<T>.Default](#). If type `TKey` implements the [System.IComparable<T>](#) generic interface, the default comparer uses that implementation. Alternatively, you can specify an implementation of the [IComparer<T>](#) generic interface by using a constructor that accepts a `comparer` parameter.

This constructor is an $O(n \log n)$ operation, where `n` is the number of elements in `dictionary`.

See also

- [IDictionary<TKey,TValue>](#)

- Default
- IComparable<T>
- IComparable

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue > (IDictionary< TKey, TValue >, IComparer< TKey >)

Initializes a new instance of the [SortedDictionary< TKey, TValue >](#) class that contains elements copied from the specified [IDictionary< TKey, TValue >](#) and uses the specified [IComparer< T >](#) implementation to compare keys.

C#

```
public SortedDictionary(System.Collections.Generic.IDictionary< TKey, TValue >
dictionary, System.Collections.Generic.IComparer< TKey >? comparer);
```

Parameters

dictionary [IDictionary< TKey, TValue >](#)

The [IDictionary< TKey, TValue >](#) whose elements are copied to the new [SortedDictionary< TKey, TValue >](#).

comparer [IComparer< TKey >](#)

The [IComparer< T >](#) implementation to use when comparing keys, or `null` to use the default [Comparer< T >](#) for the type of the key.

Exceptions

ArgumentNullException

`dictionary` is `null`.

ArgumentException

`dictionary` contains one or more duplicate keys.

Examples

The following code example shows how to use [SortedDictionary<TKey,TValue>](#) to create a case-insensitive sorted copy of the information in a case-insensitive [Dictionary<TKey,TValue>](#), by passing the [Dictionary<TKey,TValue>](#) to the [SortedDictionary<TKey,TValue>\(IDictionary<TKey,TValue>, IComparer<TKey>\)](#) constructor. In this example, the case-insensitive comparers are for the current culture.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new Dictionary of strings, with string keys and
        // a case-insensitive equality comparer for the current
        // culture.
        Dictionary<string, string> openWith =
            new Dictionary<string, string>
                (StringComparer.CurrentCultureIgnoreCase);

        // Add some elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("Bmp", "paint.exe");
        openWith.Add("DIB", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // List the contents of the Dictionary.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith)
        {
            Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
                kvp.Value);
        }

        // Create a SortedDictionary of strings with string keys and a
        // case-insensitive equality comparer for the current culture,
        // and initialize it with the contents of the Dictionary.
        SortedDictionary<string, string> copy =
            new SortedDictionary<string, string>(openWith,
```

```

        StringComparer.CurrentCultureIgnoreCase);

    // List the sorted contents of the copy.
    Console.WriteLine();
    foreach( KeyValuePair<string, string> kvp in copy )
    {
        Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
                          kvp.Value);
    }
}

/* This code example produces the following output:

Key = txt, Value = notepad.exe
Key = Bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe

Key = Bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/

```

Remarks

Every key in a [SortedDictionary<TKey,TValue>](#) must be unique according to the specified comparer; therefore, every key in the source `dictionary` must also be unique according to the specified comparer.

[SortedDictionary<TKey,TValue>](#) requires a comparer implementation to perform key comparisons. If `comparer` is `null`, this constructor uses the default generic equality comparer, [Comparer<T>.Default](#). If type `TKey` implements the [System.IComparable<T>](#) generic interface, the default comparer uses that implementation.

This constructor is an $O(n \log n)$ operation, where `n` is the number of elements in `dictionary`.

See also

- [IDictionary<TKey,TValue>](#)
- [IComparer<T>](#)
- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue>.Comparer Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the [IComparer<T>](#) used to order the elements of the [SortedDictionary<TKey,TValue>](#).

C#

```
public System.Collections.Generic.IComparer<TKey> Comparer { get; }
```

Property Value

[IComparer<TKey>](#)

The [IComparer<T>](#) used to order the elements of the [SortedDictionary<TKey,TValue>](#)

Remarks

[SortedDictionary<TKey,TValue>](#) requires a comparer implementation to perform key comparisons. You can specify an implementation of the [IComparer<T>](#) generic interface by using a constructor that accepts a `comparer` parameter. If you do not, the default generic equality comparer, [Comparer<T>.Default](#), is used. If type `TKey` implements the [System.IComparable<T>](#) generic interface, the default comparer uses that implementation.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary< TKey, TValue >](#)

SortedDictionary< TKey, TValue >.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of key/value pairs contained in the [SortedDictionary< TKey, TValue >](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of key/value pairs contained in the [SortedDictionary< TKey, TValue >](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Remarks

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey, TValue>.Item[TKey] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the value associated with the specified key.

C#

```
public TValue this[TKey key] { get; set; }
```

Parameters

key TKey

The key of the value to get or set.

Property Value

TValue

The value associated with the specified key. If the specified key is not found, a get operation throws a [KeyNotFoundException](#), and a set operation creates a new element with the specified key.

Implements

[Item\[TKey\]](#)

Exceptions

[ArgumentNullException](#)

key is **null**.

[KeyNotFoundException](#)

The property is retrieved and **key** does not exist in the collection.

Examples

The following code 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 also 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.

This code example is part of a larger example provided for the [SortedDictionary< TKey, TValue >](#) class.

C#

```
// 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";
```

C#

```
// 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.");
}
```

C#

```
// 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.");
    }
}
```

Remarks

This property provides the ability to access a specific element in the collection by using the following C# syntax: `myCollection[key]` (`myCollection(key)` in Visual Basic).

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the [SortedDictionary< TKey, TValue >](#); for example,

`myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the [SortedDictionary< TKey, TValue >](#), setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

A key cannot be `null`, but a value can be, if the value type `TValue` is a reference type.

The C# language uses the `this` keyword to define the indexers instead of implementing the [Item\[\]](#) property. Visual Basic implements [Item\[\]](#) as a default property, which provides the same indexing functionality.

Getting the value of this property is an $O(\log n)$ operation; setting the property is also an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Add\(TKey, TValue\)](#)

SortedDictionary< TKey, TValue >.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the keys in the [SortedDictionary< TKey, TValue >](#).

C#

```
public System.Collections.Generic.SortedDictionary< TKey, TValue >.KeyCollection Keys  
{ get; }
```

Property Value

[SortedDictionary< TKey, TValue >.KeyCollection](#)

A [SortedDictionary< TKey, TValue >.KeyCollection](#) containing the keys in the [SortedDictionary< TKey, TValue >](#).

Examples

The following code example shows how to enumerate the keys in the dictionary using the [Keys](#) property, and how to enumerate the keys and values in the dictionary.

This code is part of a larger example that can be compiled and executed. See [SortedDictionary< TKey, TValue >](#).

C#

```
// To get the keys alone, use the Keys property.  
SortedDictionary< 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);  
}
```

C#

```
// 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);
}
```

Remarks

The keys in the [SortedDictionary<TKey,TValue>.KeyCollection](#) are sorted according to the [Comparer](#) property and are in the same order as the associated values in the [SortedDictionary<TKey,TValue>.ValueCollection](#) returned by the [Values](#) property.

The returned [SortedDictionary<TKey,TValue>.KeyCollection](#) is not a static copy; instead, the [SortedDictionary<TKey,TValue>.KeyCollection](#) refers back to the keys in the original [SortedDictionary<TKey,TValue>](#). Therefore, changes to the [SortedDictionary<TKey,TValue>](#) continue to be reflected in the [SortedDictionary<TKey,TValue>.KeyCollection](#).

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.KeyCollection](#)
- [Values](#)

SortedDictionary< TKey, TValue >.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the values in the [SortedDictionary< TKey, TValue >](#).

C#

```
public System.Collections.Generic.SortedDictionary< TKey, TValue >.ValueCollection  
Values { get; }
```

Property Value

[SortedDictionary< TKey, TValue >.ValueCollection](#)

A [SortedDictionary< TKey, TValue >.ValueCollection](#) containing the values in the [SortedDictionary< TKey, TValue >](#).

Examples

This code example shows how to enumerate the values in the dictionary using the [Values](#) property, and how to enumerate the keys and values in the dictionary.

This code example is part of a larger example provided for the [SortedDictionary< TKey, TValue >](#) class.

C#

```
// To get the values alone, use the Values property.  
SortedDictionary< 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);  
}
```

C#

```
// 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);
}
```

Remarks

The values in the [SortedDictionary<TKey,TValue>.ValueCollection](#) are sorted according to the [Comparer](#) property, and are in the same order as the associated keys in the [SortedDictionary<TKey,TValue>.KeyCollection](#) returned by the [Keys](#) property.

The returned [SortedDictionary<TKey,TValue>.ValueCollection](#) is not a static copy; instead, the [SortedDictionary<TKey,TValue>.ValueCollection](#) refers back to the values in the original [SortedDictionary<TKey,TValue>](#). Therefore, changes to the [SortedDictionary<TKey,TValue>](#) continue to be reflected in the [SortedDictionary<TKey,TValue>.ValueCollection](#).

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.ValueCollection](#)
- [Keys](#)

SortedDictionary<TKey,TValue>.Add(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an element with the specified key and value into the [SortedDictionary<TKey,TValue>](#).

C#

```
public void Add(TKey key, TValue value);
```

Parameters

key TKey

The key of the element to add.

value TValue

The value of the element to add. The value can be `null` for reference types.

Implements

[Add\(TKey, TValue\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

An element with the same key already exists in the [SortedDictionary<TKey,TValue>](#).

Examples

The following code example creates an empty [SortedDictionary<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.

This code example is part of a larger example provided for the `SortedDictionary<TKey,TValue>` class.

C#

```
// Create a new sorted dictionary of strings, with string
// keys.
SortedDictionary<string, string> openWith =
    new SortedDictionary<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.");
}
```

Remarks

You can also use the `Item[]` property to add new elements by setting the value of a key that does not exist in the `SortedDictionary<TKey,TValue>`; for example,

`myCollection["myNonexistentKey"] = myValue` (in Visual Basic,
`myCollection("myNonexistentKey") = myValue`). However, if the specified key already exists in the `SortedDictionary<TKey,TValue>`, setting the `Item[]` property overwrites the old value. In contrast, the `Add` method throws an exception if an element with the specified key already exists.

A key cannot be `null`, but a value can be, if the value type `TValue` is a reference type.

This method is an $O(\log n)$ operation, where `n` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove\(TKey\)](#)
- [Item\[TKey\]](#)
- [Add\(TKey, TValue\)](#)

SortedDictionary<TKey,TValue>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements from the [SortedDictionary<TKey,TValue>](#).

C#

```
public void Clear();
```

Implements

[Clear\(\)](#) , [Clear\(\)](#)

Remarks

The [Count](#) property is set to 0, and references to other objects from elements of the collection are also released.

This method is an O(1) operation, since the root of the internal data structures is simply released for garbage collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue>.ContainsKey(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [SortedDictionary<TKey,TValue>](#) contains an element with the specified key.

C#

```
public bool ContainsKey(TKey key);
```

Parameters

key TKey

The key to locate in the [SortedDictionary<TKey,TValue>](#).

Returns

Boolean

`true` if the [SortedDictionary<TKey,TValue>](#) contains an element with the specified key; otherwise, `false`.

Implements

[ContainsKey\(TKey\)](#) , [ContainsKey\(TKey\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [ContainsKey](#) method to test whether a key exists prior to calling the [Add](#) method. It also shows how to use the [TryGetValue](#) method to

retrieve values, which is an efficient way to retrieve values when a program frequently tries keys that are not in the dictionary. Finally, it shows the least efficient way to test whether keys exist, by using the `Item[]` property (the indexer in C#).

This code example is part of a larger example provided for the `SortedDictionary< TKey, TValue >` class.

C#

```
// 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"]);
}
```

C#

```
// 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.");
}
```

C#

```
// 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.");
}
```

Remarks

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsValue\(TValue\)](#)

SortedDictionary< TKey, TValue >.ContainsValue(TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [SortedDictionary< TKey, TValue >](#) contains an element with the specified value.

C#

```
public bool ContainsValue(TValue value);
```

Parameters

value TValue

The value to locate in the [SortedDictionary< TKey, TValue >](#). The value can be `null` for reference types.

Returns

Boolean

`true` if the [SortedDictionary< TKey, TValue >](#) contains an element with the specified value; otherwise, `false`.

Remarks

This method determines equality using the default equality comparer [EqualityComparer< T >.Default](#) for the value type `TValue`.

This method performs a linear search; therefore, the average execution time is proportional to the [Count](#) property. That is, this method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [Default](#)

SortedDictionary< TKey, TValue >.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [SortedDictionary< TKey, TValue >](#) to the specified array of [KeyValuePair< TKey, TValue >](#) structures, starting at the specified index.

C#

```
public void CopyTo(System.Collections.Generic.KeyValuePair< TKey, TValue >[] array,  
int index);
```

Parameters

array [KeyValuePair< TKey, TValue >\[\]](#)

The one-dimensional array of [KeyValuePair< TKey, TValue >](#) structures that is the destination of the elements copied from the current [SortedDictionary< TKey, TValue >](#). The array must have zero-based indexing.

index [Int32](#)

The zero-based index in **array** at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

array is **null**.

[ArgumentOutOfRangeException](#)

index is less than 0.

[ArgumentException](#)

The number of elements in the source `SortedDictionary<TKey,TValue>` is greater than the available space from `index` to the end of the destination `array`.

Remarks

! Note

If the type of the source `SortedDictionary<TKey,TValue>` cannot be cast automatically to the type of the destination `array`, the nongeneric implementations of `ICollection.CopyTo` throw `InvalidCastException`, whereas the generic implementations throw `ArgumentException`.

This method is an $O(n)$ operation, where `n` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue>.Get Enumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [SortedDictionary<TKey,TValue>](#).

C#

```
public System.Collections.Generic.SortedDictionary<TKey,TValue>.Enumerator  
GetEnumerator();
```

Returns

[SortedDictionary<TKey,TValue>.Enumerator](#)

A [SortedDictionary<TKey,TValue>.Enumerator](#) for the [SortedDictionary<TKey,TValue>](#).

Remarks

For purposes of enumeration, each item is a [KeyValuePair<TKey,TValue>](#) structure representing a value and its key.

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

The dictionary is maintained in a sorted order using an internal tree. Every new element is positioned at the correct sort position, and the tree is adjusted to maintain the sort order whenever an element is removed. While enumerating, the sort order is maintained.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same element until the [MoveNext](#) method is called.

[MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation, where n is count.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SortedDictionary<TKey,TValue>.Enumerator](#)
- [IEnumerator<T>](#)

SortedDictionary<TKey,TValue>.Remove(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the element with the specified key from the [SortedDictionary<TKey,TValue>](#).

C#

```
public bool Remove(TKey key);
```

Parameters

key TKey

The key of the element to remove.

Returns

Boolean

`true` if the element is successfully removed; otherwise, `false`. This method also returns `false` if `key` is not found in the [SortedDictionary<TKey,TValue>](#).

Implements

[Remove\(TKey\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to remove a key/value pair from the dictionary using the [Remove](#) method.

This code example is part of a larger example provided for the [SortedDictionary<TKey,TValue>](#) class.

C#

```
// 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.");  
}
```

Remarks

If the [SortedDictionary<TKey,TValue>](#) does not contain an element with the specified key, the [SortedDictionary<TKey,TValue>](#) remains unchanged. No exception is thrown.

This method is an O(log n) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Add\(TKey, TValue\)](#)
- [Remove\(TKey\)](#)

SortedDictionary<TKey,TValue>.TryGetValue(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the value associated with the specified key.

C#

```
public bool TryGetValue(TKey key, out TValue value);
```

Parameters

key TKey

The key of the value to get.

value TValue

When this method returns, the value associated with the specified key, if the key is found; otherwise, the default value for the type of the **value** parameter.

Returns

Boolean

true if the [SortedDictionary<TKey,TValue>](#) contains an element with the specified key; otherwise, **false**.

Implements

[TryGetValue\(TKey, TValue\)](#) , [TryGetValue\(TKey, TValue\)](#)

Exceptions

[ArgumentNullException](#)

key is **null**.

Examples

The example shows how to use the [TryGetValue](#) method as a more efficient way to retrieve values in a program that frequently tries keys that are not in the dictionary. For contrast, the example also shows how the [Item\[\]](#) property (the indexer in C#) throws exceptions when attempting to retrieve nonexistent keys.

This code example is part of a larger example provided for the [SortedDictionary<TKey,TValue>](#) class.

C#

```
// 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.");
}
```

C#

```
// 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.");
}
```

Remarks

This method combines the functionality of the [ContainsKey](#) method and the [Item\[\]](#) property.

If the key is not found, then the `value` parameter gets the appropriate default value for the value type `TValue`; for example, 0 (zero) for integer types, `false` for Boolean types, and `null` for reference types.

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [Item\[TKey\]](#)

SortedDictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.Add Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to the [ICollection<T>](#).

C#

```
void  
ICollection<KeyValuePair< TKey, TValue >>.Add(System.Collections.Generic.KeyValuePair  
< TKey, TValue > keyValuePair);
```

Parameters

keyValuePair [KeyValuePair< TKey, TValue >](#)

The [KeyValuePair< TKey, TValue >](#) structure to add to the [ICollection< T >](#).

Implements

[Add\(T\)](#)

Exceptions

[ArgumentNullException](#)

`keyValuePair` is `null`.

[ArgumentException](#)

An element with the same key already exists in the [SortedDictionary< TKey, TValue >](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sorted Dictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.Contains Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [ICollection<T>](#) contains a specific key and value.

C#

```
bool  
ICollection<KeyValuePair< TKey, TValue >>.Contains(System.Collections.Generic.KeyValue  
ePair< TKey, TValue > keyValuePair);
```

Parameters

keyValuePair [KeyValuePair< TKey, TValue >](#)

The [KeyValuePair< TKey, TValue >](#) structure to locate in the [ICollection<T>](#).

Returns

[Boolean](#)

`true` if `keyValuePair` is found in the [ICollection<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [Performing Culture-Insensitive String Operations in Collections](#)

SortedDictionary< TKey, TValue >.ICollection< KeyValuePair< TKey, TValue > >.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool  
System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKe  
y, TValue>>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sorted Dictionary<TKey,TValue>.ICollection<KeyValuePair<TKey,TValue>>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of the specified element from the [ICollection<T>](#).

C#

```
bool  
ICollection<KeyValuePair<TKey,TValue>>.Remove(System.Collections.Generic.KeyValuePair<TKey,TValue> keyValuePair);
```

Parameters

keyValuePair [KeyValuePair<TKey,TValue>](#)

The [KeyValuePair<TKey,TValue>](#) structure to remove from the [ICollection<T>](#).

Returns

[Boolean](#)

`true` if `keyValuePair` was successfully removed from the [ICollection<T>](#); otherwise, `false`. This method also returns `false` if `keyValuePair` was not found in the [ICollection<T>](#).

Implements

[Remove\(T\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sorted Dictionary< TKey, TValue >. IDictionary< TKey, TValue >. Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection<T>](#) containing the keys of the [IDictionary< TKey, TValue >](#).

C#

```
System.Collections.Generic.ICollection<TKey>
System.Collections.Generic.IDictionary< TKey, TValue >.Keys { get; }
```

Property Value

[ICollection< TKey >](#)

An [ICollection<T>](#) containing the keys of the [IDictionary< TKey, TValue >](#).

Implements

[Keys](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sorted Dictionary< TKey, TValue >. IDictionary< TKey, TValue >. Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection<T>](#) containing the values in the [IDictionary< TKey, TValue >](#).

C#

```
System.Collections.Generic.ICollection<TValue>
System.Collections.Generic.IDictionary< TKey, TValue >.Values { get; }
```

Property Value

[ICollection< TValue >](#)

An [ICollection< T >](#) containing the values in the [IDictionary< TKey, TValue >](#).

Implements

[Values](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>> IEnumerable<KeyValuePair<TKey, TValue>>.GetEnumerator();
```

Returns

[IEnumerator<KeyValuePair<TKey, TValue>>](#)

An enumerator that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue>.IReadOnlyDictionary<TKey,TValue>.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the keys in the [SortedDictionary<TKey,TValue>](#).

C#

```
System.Collections.Generic.IEnumerable<TKey>
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>.Keys { get; }
```

Property Value

[IEnumerable<TKey>](#)

A collection containing the keys in the [SortedDictionary<TKey,TValue>](#).

Implements

[Keys](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue>.IReadOnlyDictionary<TKey,TValue>.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the values in the [SortedDictionary<TKey,TValue>](#).

C#

```
System.Collections.Generic.IEnumerable<TValue>
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>.Values { get; }
```

Property Value

[IEnumerable<TValue>](#)

A collection containing the values in the [SortedDictionary<TKey,TValue>](#).

Implements

[Values](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sorted Dictionary< TKey, TValue >.ICollection.CopyTo To(Array, Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection<T>](#) to an array, starting at the specified array index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional array that is the destination of the elements copied from the [ICollection<T>](#). The array must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than 0.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source `ICollection<T>` is greater than the available space from `index` to the end of the destination `array`.

-or-

The type of the source `ICollection<T>` cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source `ICollection` cannot be cast automatically to the type of the destination `array`, the nongeneric implementations of `ICollection.CopyTo` throw an `InvalidCastException`, whereas the generic implementations throw an `ArgumentException`.

This method is an $O(n)$ operation, where `n` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. Even when a collection is synchronized, other threads can still modify the collection, which can cause the enumerator to throw an exception. To guarantee thread safety during enumeration, you can either lock the collection during the entire enumeration or catch the exceptions resulting from changes made by other threads.

The [ICollection.SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

Sorted Dictionary< TKey, TValue >.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#).

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [ICollection.SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock the object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;
lock (ic.SyncRoot)
{
    // Access the collection.
}
```

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

Sorted Dictionary<TKey,TValue>.IDictionary.Add(Object, Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an element with the provided key and value to the [IDictionary](#).

C#

```
void IDictionary.Add(object key, object value);
```

Parameters

key [Object](#)

The object to use as the key of the element to add.

value [Object](#)

The object to use as the value of the element to add.

Implements

[Add\(Object, Object\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

`key` is of a type that is not assignable to the key type `TKey` of the [IDictionary](#).

-or-

`value` is of a type that is not assignable to the value type `TValue` of the [IDictionary](#).

-or-

An element with the same key already exists in the [IDictionary](#).

Examples

The following code example shows how to access the `SortedDictionary<TKey,TValue>` class through the [System.Collections.IDictionary](#) interface. The code example creates an empty `SortedDictionary<TKey,TValue>` of strings with string keys and uses the [IDictionary.Add](#) method to add some elements. The example demonstrates that the [IDictionary.Add](#) method throws an [ArgumentException](#) when attempting to add a duplicate key, or when a key or value of the wrong data type is supplied.

The code example demonstrates the use of several other members of the [System.Collections.IDictionary](#) interface.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //

        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
        try
        {
            openWith.Add(42, new Example());
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("An exception was caught for " +
                "IDictionary.Add. Exception message:\n\t{0}\n",
                ex.Message);
        }

        // 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 returns null if the key is of the wrong data
// type.
Console.WriteLine("The indexer returns null"
    + " if the key is of the wrong type:");
Console.WriteLine("For key = 2, value = {0}.",
    openWith[2]);

// The indexer throws an exception when setting a value
// if the key is of the wrong data type.
try
{
    openWith[2] = "This does not get added.";
}
catch (ArgumentException)
{
    Console.WriteLine("A key of the wrong type was specified"
        + " when assigning to the indexer.");
}

// Unlike the default Item property on the Dictionary class
// itself, IDictionary.Item does not throw an exception
// if the requested key is not in the dictionary.
Console.WriteLine("For key = \"tif\", value = {0}.",
    openWith["tif"]);

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

```
        openWith["ht"]);
    }

    // IDictionary.Contains returns false if the wrong data
    // type is supplied.
    Console.WriteLine("openWith.Contains(29.7) returns {0}",
        openWith.Contains(29.7));

    // When you use foreach to enumerate dictionary elements
    // with the IDictionary interface, the elements are retrieved
    // as DictionaryEntry objects instead of 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 icoll = openWith.Values;

    // The elements of the collection are strongly typed
    // with the type that was specified for dictionary values,
    // even though the ICollection interface is not strongly
    // typed.
    Console.WriteLine();
    foreach( string s in icoll )
    {
        Console.WriteLine("Value = {0}", s);
    }

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

    // The elements of the collection are strongly typed
    // with the type that was specified for dictionary keys,
    // even though the ICollection interface is not strongly
    // typed.
    Console.WriteLine();
    foreach( string s in icoll )
    {
        Console.WriteLine("Key = {0}", s);
    }

    // Use the Remove method to remove a key/value pair. No
    // exception is thrown if the wrong data type is supplied.
    Console.WriteLine("\nRemove(\"dib\")");
    openWith.Remove("dib");

    if (!openWith.Contains("dib"))
    {
        Console.WriteLine("Key \"dib\" is not found.");
    }
}
```

```

/* This code example produces the following output:

An exception was caught for IDictionary.Add. Exception message:
    The value "42" is not of type "System.String" and cannot be used in this
generic collection.
Parameter name: key

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
The indexer returns null if the key is of the wrong type:
For key = 2, value = .
A key of the wrong type was specified when assigning to the indexer.
For key = "tif", value = .
Value added for key = "ht": hypertrm.exe
openWith.Contains(29.7) returns False

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

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

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

Remove("dib")
Key "dib" is not found.
*/

```

Remarks

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the dictionary, setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

This method is an $O(\log n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Item\[Object\]](#)

SortedDictionary< TKey, TValue >.IDictionary.Contains(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [IDictionary](#) contains an element with the specified key.

C#

```
bool IDictionary.Contains(object key);
```

Parameters

key [Object](#)

The key to locate in the [IDictionary](#).

Returns

[Boolean](#)

`true` if the [IDictionary](#) contains an element with the key; otherwise, `false`.

Implements

[Contains\(Object\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [IDictionary.Contains](#) method of the [System.Collections.IDictionary](#) interface with a [SortedDictionary< TKey, TValue >](#). The example

demonstrates that the method returns `false` if a key of the wrong data type is supplied.

The code example is part of a larger example, including output, provided for the `IDictionary.Add` method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
```

C#

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

// IDictionary.Contains returns false if the wrong data
// type is supplied.
Console.WriteLine("openWith.Contains(29.7) returns {0}",
                  openWith.Contains(29.7));
```

C#

```
}
```

Remarks

This method returns `false` if `key` is of a type that is not assignable to the key type `TKey` of the `SortedDictionary<TKey,TValue>`.

This method is an $O(\log n)$ operation, where `n` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.IDictionary.Get Enumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an [IDictionaryEnumerator](#) for the [IDictionary](#).

C#

```
System.Collections.IDictionaryEnumerator IDictionary.GetEnumerator();
```

Returns

[IDictionaryEnumerator](#)

An [IDictionaryEnumerator](#) for the [IDictionary](#).

Implements

[GetEnumerator\(\)](#)

Examples

The following code example shows how to enumerate the key/value pairs in the dictionary by using the `foreach` statement (`For Each` in Visual Basic), which hides the use of the enumerator. In particular, note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

```
public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
    }
}
```

C#

```
// When you use foreach to enumerate dictionary elements
// with the IDictionary interface, the elements are retrieved
// as DictionaryEntry objects instead of KeyValuePair objects.
Console.WriteLine();
foreach( DictionaryEntry de in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
        de.Key, de.Value);
}
```

C#

```
}
```

Remarks

For purposes of enumeration, each item is a [DictionaryEntry](#) structure.

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. The [Reset](#) method also brings the enumerator back to this position. At this position, [Entry](#) is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Entry](#).

The [Entry](#) property returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Entry](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Entry](#) is undefined. To set [Entry](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IDictionaryEnumerator](#)
- [IEnumerator](#)

SortedDictionary< TKey, TValue >.IDictionary.IsFixedSize Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [IDictionary](#) has a fixed size.

C#

```
bool System.Collections.IDictionary.IsFixedSize { get; }
```

Property Value

[Boolean](#)

`true` if the [IDictionary](#) has a fixed size; otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >](#), this property always returns `false`.

Implements

[IsFixedSize](#)

Remarks

A collection with a fixed size does not allow the addition or removal of elements after the collection is created, but it allows the modification of existing elements.

A collection with a fixed size is simply a collection with a wrapper that prevents adding and removing elements; therefore, if changes are made to the underlying collection, including the addition or removal of elements, the fixed-size collection reflects those changes.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.IDictionary.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [IDictionary](#) is read-only.

C#

```
bool System.Collections.IDictionary.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [IDictionary](#) is read-only; otherwise, `false`. In the default implementation of [SortedDictionary< TKey, TValue >](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.IDictionary. Item[Object] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the element with the specified key.

C#

```
object? System.Collections.IDictionary.Item[object key] { get; set; }
```

Parameters

key [Object](#)

The key of the element to get.

Property Value

[Object](#)

The element with the specified key, or `null` if `key` is not in the dictionary or `key` is of a type that is not assignable to the key type `TKey` of the [SortedDictionary< TKey, TValue >](#).

Implements

[Item\[Object\]](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

A value is being assigned, and `key` is of a type that is not assignable to the key type `TKey` of the [SortedDictionary< TKey, TValue >](#).

-or-

A value is being assigned and is of a type that isn't assignable to the value type `TValue` of the `SortedDictionary<TKey,TValue>`.

Examples

The following code example shows how to use the `IDictionary.Item[]` property (the indexer in C#) of the `System.Collections.IDictionary` interface with a `SortedDictionary<TKey,TValue>`, and ways the property differs from the `SortedDictionary<TKey,TValue>.Item[]` property.

The example shows that, like the `SortedDictionary<TKey,TValue>.Item[]` property, the `SortedDictionary<TKey,TValue>.IDictionary.Item[]` property can change the value associated with an existing key and can be used to add a new key/value pair if the specified key is not in the dictionary. The example also shows that unlike the `SortedDictionary<TKey,TValue>.Item[]` property, the `SortedDictionary<TKey,TValue>.IDictionary.Item[]` property does not throw an exception if `key` is not in the dictionary, returning a null reference instead. Finally, the example demonstrates that getting the `SortedDictionary<TKey,TValue>.IDictionary.Item[]` property returns a null reference if `key` is not the correct data type, and that setting the property throws an exception if `key` is not the correct data type.

The code example is part of a larger example, including output, provided for the `IDictionary.Add` method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
```

```
openWith.Add("dib", "paint.exe");
openWith.Add("rtf", "wordpad.exe");
```

C#

```
// 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 returns null if the key is of the wrong data
// type.
Console.WriteLine("The indexer returns null"
    + " if the key is of the wrong type:");
Console.WriteLine("For key = 2, value = {0}.",
    openWith[2]);

// The indexer throws an exception when setting a value
// if the key is of the wrong data type.
try
{
    openWith[2] = "This does not get added.";
}
catch (ArgumentException)
{
    Console.WriteLine("A key of the wrong type was specified"
        + " when assigning to the indexer.");
}
```

C#

```
// Unlike the default Item property on the Dictionary class
// itself, IDictionary.Item does not throw an exception
// if the requested key is not in the dictionary.
Console.WriteLine("For key = \"tif\", value = {0}.",
    openWith["tif"]);
```

C#

```
}
```

```
}
```

Remarks

This property provides the ability to access a specific element in the collection by using the following C# syntax: `myCollection[key]` (`myCollection(key)` in Visual Basic).

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the dictionary, setting the [Item\[\]](#) property overwrites the old value. In contrast, the [IDictionary.Add](#) method does not modify existing elements.

The C# language uses the `this` keyword to define the indexers instead of implementing the [IDictionary.Item\[\]](#) property. Visual Basic implements [IDictionary.Item\[\]](#) as a default property, which provides the same indexing functionality.

Getting the value of this property is an $O(\log n)$ operation; setting the property is also an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Add\(Object, Object\)](#)

SortedDictionary< TKey, TValue >.IDictionary.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection](#) containing the keys of the [IDictionary](#).

C#

```
System.Collections.ICollection System.Collections.IDictionary.Keys { get; }
```

Property Value

[ICollection](#)

An [ICollection](#) containing the keys of the [IDictionary](#).

Implements

[Keys](#)

Examples

The following code example shows how to use the [Keys](#) property of the [System.Collections.IDictionary](#) interface with a [SortedDictionary< TKey, TValue >](#), to list the keys in the dictionary. The example also shows how to enumerate the key/value pairs in the dictionary; note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

```
public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
    }
}
```

C#

```
// To get the keys alone, use the Keys property.
icoll = openWith.Keys;

// The elements of the collection are strongly typed
// with the type that was specified for dictionary keys,
// even though the ICollection interface is not strongly
// typed.
Console.WriteLine();
foreach( string s in icoll )
{
    Console.WriteLine("Key = {0}", s);
}
```

C#

```
// When you use foreach to enumerate dictionary elements
// with the IDictionary interface, the elements are retrieved
// as DictionaryEntry objects instead of KeyValuePair objects.
Console.WriteLine();
foreach( DictionaryEntry de in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
        de.Key, de.Value);
}
```

C#

```
}
```

Remarks

The keys in the returned [ICollection](#) are sorted according to the [Comparer](#) property and are guaranteed to be in the same order as the corresponding values in the [ICollection](#) returned by the [Values](#) property.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)

SortedDictionary< TKey, TValue >.IDictionary.Remove(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the element with the specified key from the [IDictionary](#).

C#

```
void IDictionary.Remove(object key);
```

Parameters

key [Object](#)

The key of the element to remove.

Implements

[Remove\(Object\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [IDictionary.Remove](#) of the [System.Collections.IDictionary](#) interface with a [SortedDictionary< TKey, TValue >](#).

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
    }
}
```

C#

```
// Use the Remove method to remove a key/value pair. No
// exception is thrown if the wrong data type is supplied.
Console.WriteLine("\nRemove(\"dib\")");
openWith.Remove("dib");

if (!openWith.Contains("dib"))
{
    Console.WriteLine("Key \"dib\" is not found.");
}
```

C#

```
}
```

Remarks

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.IDictionary.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection](#) containing the values in the [IDictionary](#).

C#

```
System.Collections.ICollection System.Collections.IDictionary.Values { get; }
```

Property Value

[ICollection](#)

An [ICollection](#) containing the values in the [IDictionary](#).

Implements

[Values](#)

Examples

The following code example shows how to use the [IDictionary.Values](#) property of the [System.Collections.IDictionary](#) interface with a [SortedDictionary< TKey, TValue >](#), to list the values in the dictionary. The example also shows how to enumerate the key/value pairs in the dictionary; note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

```
public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
    }
}
```

C#

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

// The elements of the collection are strongly typed
// with the type that was specified for dictionary values,
// even though the ICollection interface is not strongly
// typed.
Console.WriteLine();
foreach( string s in icoll )
{
    Console.WriteLine("Value = {0}", s);
}
```

C#

```
// When you use foreach to enumerate dictionary elements
// with the IDictionary interface, the elements are retrieved
// as DictionaryEntry objects instead of KeyValuePair objects.
Console.WriteLine();
foreach( DictionaryEntry de in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
                     de.Key, de.Value);
}
```

C#

```
}
```

Remarks

The values in the returned [ICollection](#) are sorted according to the [Comparer](#) property, and are guaranteed to be in the same order as the corresponding keys in the [ICollection](#) returned by the [Keys](#) property.

Getting the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)

SortedDictionary< TKey, TValue >. IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

The [Current](#) property returns the same element until the [MoveNext](#) method is called.

[MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

This method is an $O(\log n)$ operation where n is a number of elements in a collection.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetEnumerator\(\)](#)
- [IEnumerator<T>](#)

SortedList<TKey,TValue> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a collection of key/value pairs that are sorted by key based on the associated [IComparer<T>](#) implementation.

C#

```
public class SortedList<TKey, TValue> :  
    System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>, System.Collections.Generic.IDictionary<TKey, TValue>,  
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>>,  
    System.Collections.Generic.IReadOnlyCollection<System.Collections.Generic.KeyValuePair<TKey, TValue>>, System.Collections.Generic.IReadOnlyDictionary<TKey, TValue>,  
    System.Collections.IDictionary
```

Type Parameters

TKey

The type of keys in the collection.

TValue

The type of values in the collection.

Inheritance [Object](#) → SortedList<TKey, TValue>

Derived [System.IdentityModel.Metadata.IndexedProtocolEndpointDictionary](#)

Implements [ICollection<KeyValuePair<TKey, TValue>>](#) , [IDictionary<TKey, TValue>](#) ,
[IEnumerable<KeyValuePair<TKey, TValue>>](#) , [IEnumerable<T>](#) ,
[IReadOnlyCollection<KeyValuePair<TKey, TValue>>](#) ,
[IReadOnlyDictionary<TKey, TValue>](#) , [ICollection](#) , [IDictionary](#) , [IEnumerable](#)

Examples

The following code example creates an empty `SortedList<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 sorted list, 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 sorted list and how to enumerate the keys and values alone using the `Keys` property and the `Values` property.

Finally, the example demonstrates the `Remove` method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string
        // keys.
        SortedList<string, string> openWith =
            new SortedList<string, string>();

        // Add some elements to the list. 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 list.
        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 list.
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 list, 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 list 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.
IList<string> ilistValues = openWith.Values;

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

// The Values property is an efficient way to retrieve
// values by index.
Console.WriteLine("\nIndexed retrieval using the Values " +
    "property: Values[2] = {0}", openWith.Values[2]);

// To get the keys alone, use the Keys property.
IList<string> ilistKeys = openWith.Keys;

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

// The Keys property is an efficient way to retrieve
// keys by index.
Console.WriteLine("\nIndexed retrieval using the Keys " +
    "property: Keys[2] = {0}", openWith.Keys[2]);

// 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 = bmp, Value = paint.exe
Key = dib, Value = paint.exe

```

```
Key = doc, Value = winword.exe  
Key = ht, Value = hypertrm.exe  
Key = rtf, Value = winword.exe  
Key = txt, Value = notepad.exe
```

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

Indexed retrieval using the `Values` property: `Values[2] = winword.exe`

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

Indexed retrieval using the `Keys` property: `Keys[2] = doc`

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

Remarks

The `SortedList<TKey,TValue>` generic class is an array of key/value pairs with $O(\log n)$ retrieval, where n is the number of elements in the dictionary. In this, it is similar to the `SortedDictionary<TKey,TValue>` generic class. The two classes have similar object models, and both have $O(\log n)$ retrieval. Where the two classes differ is in memory use and speed of insertion and removal:

- `SortedList<TKey,TValue>` uses less memory than `SortedDictionary<TKey,TValue>`.
- `SortedDictionary<TKey,TValue>` has faster insertion and removal operations for unsorted data, $O(\log n)$ as opposed to $O(n)$ for `SortedList<TKey,TValue>`.
- If the list is populated all at once from sorted data, `SortedList<TKey,TValue>` is faster than `SortedDictionary<TKey,TValue>`.

Another difference between the `SortedDictionary<TKey,TValue>` and `SortedList<TKey,TValue>` classes is that `SortedList<TKey,TValue>` supports efficient indexed retrieval of keys and values through the collections returned by the `Keys` and `Values` properties. It is not necessary to regenerate the lists when the properties are accessed, because the lists are just wrappers for

the internal arrays of keys and values. The following code shows the use of the `Values` property for indexed retrieval of values from a sorted list of strings:

```
C#
```

```
string v = mySortedList.Values[3];
```

`SortedList<TKey,TValue>` is implemented as an array of key/value pairs, sorted by the key. Each element can be retrieved as a `KeyValuePair<TKey,TValue>` object.

Key objects must be immutable as long as they are used as keys in the `SortedList<TKey,TValue>`. Every key in a `SortedList<TKey,TValue>` must be unique. A key cannot be `null`, but a value can be, if the type of values in the list, `TValue`, is a reference type.

`SortedList<TKey,TValue>` requires a comparer implementation to sort and to perform comparisons. The default comparer `Comparer<T>.Default` checks whether the key type `TKey` implements `System.IComparable<T>` and uses that implementation, if available. If not, `Comparer<T>.Default` checks whether the key type `TKey` implements `System.IComparable`. If the key type `TKey` does not implement either interface, you can specify a `System.Collections.Generic.IComparer<T>` implementation in a constructor overload that accepts a `comparer` parameter.

The capacity of a `SortedList<TKey,TValue>` is the number of elements the `SortedList<TKey,TValue>` can hold. As elements are added to a `SortedList<TKey,TValue>`, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling `TrimExcess` or by setting the `Capacity` property explicitly. Decreasing the capacity reallocates memory and copies all the elements in the `SortedList<TKey,TValue>`.

.NET Framework only: For very large `SortedList<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 `<gcAllowVeryLargeObjects>` 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 the elements of the `SortedList<TKey,TValue>` are key/value pairs, the element type is not the type of the key or the type of the value. Instead, the element type is `KeyValuePair<TKey,TValue>`. For example:

```
C#
```

```
foreach( KeyValuePair<int, string> kvp in mySortedList )  
{
```

```
        Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
    }
```

The `foreach` statement is a wrapper around the enumerator, which only allows reading from, not writing to, the collection.

Constructors

[+] Expand table

<code>SortedList<TKey,TValue>()</code>	Initializes a new instance of the <code>SortedList<TKey,TValue></code> class that is empty, has the default initial capacity, and uses the default <code>IComparer<T></code> .
<code>SortedList<TKey,TValue>(IComparer<TKey>)</code>	Initializes a new instance of the <code>SortedList<TKey,TValue></code> class that is empty, has the default initial capacity, and uses the specified <code>IComparer<T></code> .
<code>SortedList<TKey,TValue>(IDictionary<TKey,TValue>, IComparer<TKey>)</code>	Initializes a new instance of the <code>SortedList<TKey,TValue></code> class that contains elements copied from the specified <code>IDictionary<TKey,TValue></code> , has sufficient capacity to accommodate the number of elements copied, and uses the specified <code>IComparer<T></code> .
<code>SortedList<TKey,TValue>(IDictionary<TKey,TValue>)</code>	Initializes a new instance of the <code>SortedList<TKey,TValue></code> class that contains elements copied from the specified <code>IDictionary<TKey,TValue></code> , has sufficient capacity to accommodate the number of elements copied, and uses the default <code>IComparer<T></code> .
<code>SortedList<TKey,TValue>(Int32, IComparer<TKey>)</code>	Initializes a new instance of the <code>SortedList<TKey,TValue></code> class that is empty, has the specified initial capacity, and uses the specified <code>IComparer<T></code> .
<code>SortedList<TKey,TValue>(Int32)</code>	Initializes a new instance of the <code>SortedList<TKey,TValue></code> class that is empty, has the specified initial capacity, and uses the default <code>IComparer<T></code> .

Properties

[+] Expand table

<code>Capacity</code>	Gets or sets the number of elements that the <code>SortedList<TKey,TValue></code> can contain.
<code>Comparer</code>	Gets the <code>IComparer<T></code> for the sorted list.

Count	Gets the number of key/value pairs contained in the SortedList<TKey,TValue> .
Item[TKey]	Gets or sets the value associated with the specified key.
Keys	Gets a collection containing the keys in the SortedList<TKey,TValue> , in sorted order.
Values	Gets a collection containing the values in the SortedList<TKey,TValue> .

Methods

 [Expand table](#)

Add(TKey, TValue)	Adds an element with the specified key and value into the SortedList<TKey,TValue> .
Clear()	Removes all elements from the SortedList<TKey,TValue> .
ContainsKey(TKey)	Determines whether the SortedList<TKey,TValue> contains a specific key.
ContainsValue(TValue)	Determines whether the SortedList<TKey,TValue> contains a specific value.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the SortedList<TKey,TValue> .
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
IndexOfKey(TKey)	Searches for the specified key and returns the zero-based index within the entire SortedList<TKey,TValue> .
IndexOfValue(TValue)	Searches for the specified value and returns the zero-based index of the first occurrence within the entire SortedList<TKey,TValue> .
MemberwiseClone()	Creates a shallow copy of the current Object . (Inherited from Object)
Remove(TKey)	Removes the element with the specified key from the SortedList<TKey,TValue> .
RemoveAt(Int32)	Removes the element at the specified index of the SortedList<TKey,TValue> .
ToString()	Returns a string that represents the current object. (Inherited from Object)

TrimExcess()	Sets the capacity to the actual number of elements in the SortedList<TKey,TValue> , if that number is less than 90 percent of current capacity.
TryGetValue(TKey, TValue)	Gets the value associated with the specified key.

Explicit Interface Implementations

[\[+\] Expand table](#)

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array , starting at a particular Array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
ICollection<KeyValuePair<TKey,TValue>>.Add(KeyValuePair<TKey,TValue>)	Adds a key/value pair to the ICollection<T> .
ICollection<KeyValuePair<TKey,TValue>>.Contains(KeyValuePair<TKey,TValue>)	Determines whether the ICollection<T> contains a specific element.
ICollection<KeyValuePair<TKey,TValue>>.CopyTo(KeyValuePair<TKey,TValue>[], Int32)	Copies the elements of the ICollection<T> to an Array , starting at a particular Array index.
ICollection<KeyValuePair<TKey,TValue>>.IsReadOnly	Gets a value indicating whether the ICollection<T> is read-only.
ICollection<KeyValuePair<TKey,TValue>>.Remove(KeyValuePair<TKey,TValue>)	Removes the first occurrence of a specific key/value pair from the ICollection<T> .
IDictionary.Add(Object, Object)	Adds an element with the provided key and value to the IDictionary .
IDictionary.Contains(Object)	Determines whether the IDictionary contains an element with the specified key.
IDictionary.GetEnumerator()	Returns an IDictionaryEnumerator for the IDictionary .
IDictionary.IsFixedSize	Gets a value indicating whether the IDictionary has a fixed size.

IDictionary.IsReadOnly	Gets a value indicating whether the IDictionary is read-only.
IDictionary.Item[Object]	Gets or sets the element with the specified key.
IDictionary.Keys	Gets an ICollection containing the keys of the IDictionary .
IDictionary.Remove(Object)	Removes the element with the specified key from the IDictionary .
IDictionary.Values	Gets an ICollection containing the values in the IDictionary .
IDictionary< TKey, TValue >.Keys	Gets an ICollection< T > containing the keys of the IDictionary< TKey, TValue > .
IDictionary< TKey, TValue >.Values	Gets an ICollection< T > containing the values in the IDictionary< TKey, TValue > .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through a collection.
IEnumerable< KeyValuePair< TKey, TValue > >.Get Enumerator()	Returns an enumerator that iterates through a collection.
IReadOnlyDictionary< TKey, TValue >.Keys	Gets an enumerable collection that contains the keys in the read-only dictionary.
IReadOnlyDictionary< TKey, TValue >.Values	Gets an enumerable collection that contains the values in the read-only dictionary.

Extension Methods

[+] [Expand table](#)

GetValueOrDefault< TKey, TValue > (IReadOnly Dictionary< TKey, TValue >, TKey, TValue)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
GetValueOrDefault< TKey, TValue > (IReadOnly Dictionary< TKey, TValue >, TKey)	Tries to get the value associated with the specified <code>key</code> in the <code>dictionary</code> .
Remove< TKey, TValue > (IDictionary< TKey, TValue >, TKey, TValue)	Tries to remove the value with the specified <code>key</code> from the <code>dictionary</code> .

TryAdd<TKey,TValue>(IDictionary<TKey,TValue>, TKey, TValue)	Tries to add the specified <code>key</code> and <code>value</code> to the <code>dictionary</code> .
ToImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
ToImmutableDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
ToImmutableHashSet<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable hash set of its contents.
ToImmutableList<TSource>(IEnumerable<TSource>)	Enumerates a sequence and produces an immutable list of its contents.
ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.

<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.

All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether all elements of a sequence satisfy a condition.
Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether any element of a sequence satisfies a condition.
Any<TSource>(IEnumerable<TSource>)	Determines whether a sequence contains any elements.
Append<TSource>(IEnumerable<TSource>, TSource)	Appends a value to the end of the sequence.
AsEnumerable<TSource>(IEnumerable<TSource>)	Returns the input typed as IEnumerable<T> .
Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the average of a sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the average of a sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the average of a sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on

	each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in

	a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.

<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key

<code>Comparer<TKey></code>	selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>,</code>	Correlates the elements of two sequences based on equality of

<code>Func<TOuter, IEnumerable<TInner>, TResult>()</code>	keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.

<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.

<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a

	<p>result selector function on each element therein.</p>
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	<p>Projects each element of a sequence to an IEnumerable<T>, flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	<p>Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T>.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	<p>Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.</p>
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.</p>
Single<TSource>(IEnumerable<TSource>)	<p>Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	<p>Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	<p>Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified</p>

	condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform

	function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Computes the sum of the sequence of nullable <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Computes the sum of the sequence of nullable <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Computes the sum of the sequence of <code>Single</code> values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.

<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToList<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .

ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an IEnumerable to an IQueryable .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of every node in the source collection.
<code>DescendantNodes<T>(IEnumerable<T>)</code>	Returns a collection of the descendant nodes of every document and element in the source collection.
<code>Descendants<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
<code>Descendants<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the descendant

	elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [SortedList<TKey,TValue>](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during

the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

See also

- [IDictionary< TKey, TValue >](#)
- [Dictionary< TKey, TValue >](#)
- [SortedDictionary< TKey, TValue >](#)
- [KeyValuePair< TKey, TValue >](#)
- [IComparer< T >](#)

SortedList<TKey,TValue> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [SortedList<TKey,TValue>](#) class.

Overloads

[+] [Expand table](#)

SortedList<TKey,TValue>()	Initializes a new instance of the SortedList<TKey,TValue> class that is empty, has the default initial capacity, and uses the default IComparer<T> .
SortedList<TKey,TValue>(IComparer<TKey>)	Initializes a new instance of the SortedList<TKey,TValue> class that is empty, has the default initial capacity, and uses the specified IComparer<T> .
SortedList<TKey,TValue>(IDictionary<TKey,TValue>)	Initializes a new instance of the SortedList<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> , has sufficient capacity to accommodate the number of elements copied, and uses the default IComparer<T> .
SortedList<TKey,TValue>(Int32)	Initializes a new instance of the SortedList<TKey,TValue> class that is empty, has the specified initial capacity, and uses the default IComparer<T> .
SortedList<TKey,TValue>(IDictionary<TKey,TValue>, IComparer<TKey>)	Initializes a new instance of the SortedList<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> , has sufficient capacity to accommodate the number of elements copied, and uses the specified IComparer<T> .
SortedList<TKey,TValue>(Int32, IComparer<TKey>)	Initializes a new instance of the SortedList<TKey,TValue> class that is empty, has the specified initial capacity, and uses the specified IComparer<T> .

SortedList<TKey,TValue>()

Initializes a new instance of the [SortedList<TKey,TValue>](#) class that is empty, has the default initial capacity, and uses the default [IComparer<T>](#).

C#

```
public SortedList();
```

Examples

The following code example creates an empty [SortedList<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.

This code example is part of a larger example provided for the [SortedList<TKey,TValue>](#) class.

C#

```
// Create a new sorted list of strings, with string
// keys.
SortedList<string, string> openWith =
    new SortedList<string, string>();

// Add some elements to the list. 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 list.
try
{
    openWith.Add("txt", "winword.exe");
}
catch (ArgumentException)
{
    Console.WriteLine("An element with Key = \"txt\" already exists.");
}
```

Remarks

Every key in a [SortedList<TKey,TValue>](#) must be unique according to the default comparer.

This constructor uses the default value for the initial capacity of the [SortedList<TKey,TValue>](#). To set the initial capacity, use the [SortedList<TKey,TValue>\(Int32\)](#) constructor. If the final size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [SortedList<TKey,TValue>](#).

This constructor uses the default comparer for `TKey`. To specify a comparer, use the [SortedList<TKey,TValue>\(IComparer<TKey>\)](#) constructor. The default comparer `Comparer<T>.Default` checks whether the key type `TKey` implements [System.IComparable<T>](#) and uses that implementation, if available. If not, `Comparer<T>.Default` checks whether the key type `TKey` implements [System.IComparable](#). If the key type `TKey` does not implement either interface, you can specify a [System.Collections.Generic.IComparer<T>](#) implementation in a constructor overload that accepts a `comparer` parameter.

This constructor is an O(1) operation.

See also

- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedList<TKey,TValue>(IComparer<TKey>)

Initializes a new instance of the [SortedList<TKey,TValue>](#) class that is empty, has the default initial capacity, and uses the specified [IComparer<T>](#).

C#

```
public SortedList(System.Collections.Generic.IComparer<TKey>? comparer);
```

Parameters

comparer `IComparer<TKey>`

The `IComparer<T>` implementation to use when comparing keys.

-or-

`null` to use the default `Comparer<T>` for the type of the key.

Examples

The following code example creates a sorted list with a case-insensitive comparer for the current culture. The example adds four elements, some with lower-case keys and some with upper-case keys. The example then attempts to add an element with a key that differs from an existing key only by case, catches the resulting exception, and displays an error message. Finally, the example displays the elements in case-insensitive sort order.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string keys and
        // a case-insensitive comparer for the current culture.
        SortedList<string, string> openWith =
            new SortedList<string, string>(
                StringComparer.CurrentCultureIgnoreCase);

        // Add some elements to the list.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("DIB", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Try to add a fifth element with a key that is the same
        // except for case; this would be allowed with the default
        // comparer.
        try
        {
            openWith.Add("BMP", "paint.exe");
        }
    }
}
```

```

    }
    catch (ArgumentException)
    {
        Console.WriteLine("\nBMP is already in the sorted list.");
    }

    // List the contents of the sorted list.
    Console.WriteLine();
    foreach( KeyValuePair<string, string> kvp in openWith )
    {
        Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
            kvp.Value);
    }
}

/* This code example produces the following output:

BMP is already in the sorted list.

Key = bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/

```

Remarks

Every key in a [SortedList<TKey,TValue>](#) must be unique according to the specified comparer.

This constructor uses the default value for the initial capacity of the [SortedList<TKey,TValue>](#). To set the initial capacity, use the [SortedList<TKey,TValue>\(Int32, IComparer<TKey>\)](#) constructor. If the final size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [SortedList<TKey,TValue>](#).

This constructor is an O(1) operation.

See also

- [IComparer<T>](#)
- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey,TValue> (IDictionary<TKey,TValue>)

Initializes a new instance of the [SortedDictionary<TKey,TValue>](#) class that contains elements copied from the specified [IDictionary<TKey,TValue>](#), has sufficient capacity to accommodate the number of elements copied, and uses the default [IComparer<T>](#).

C#

```
public SortedDictionary(System.Collections.Generic.IDictionary<TKey, TValue>
dictionary);
```

Parameters

dictionary [IDictionary<TKey,TValue>](#)

The [IDictionary<TKey,TValue>](#) whose elements are copied to the new [SortedDictionary<TKey,TValue>](#).

Exceptions

[ArgumentNullException](#)

`dictionary` is `null`.

[ArgumentException](#)

`dictionary` contains one or more duplicate keys.

Examples

The following code example shows how to use `SortedList< TKey, TValue >` to create a sorted copy of the information in a `Dictionary< TKey, TValue >`, by passing the `Dictionary< TKey, TValue >` to the `SortedList< TKey, TValue >(IDictionary< TKey, TValue >)` constructor.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new Dictionary of strings, with string keys.
        //
        Dictionary<string, string> openWith =
            new Dictionary<string, string>();

        // Add some elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Create a SortedList of strings with string keys,
        // and initialize it with the contents of the Dictionary.
        SortedList<string, string> copy =
            new SortedList<string, string>(openWith);

        // List the contents of the copy.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in copy )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }

    /* This code example produces the following output:

    Key = bmp, Value = paint.exe
    Key = dib, Value = paint.exe
    Key = rtf, Value = wordpad.exe
    Key = txt, Value = notepad.exe
    */
}
```

Remarks

Every key in a [SortedList<TKey,TValue>](#) must be unique according to the default comparer; likewise, every key in the source `dictionary` must also be unique according to the default comparer.

The capacity of the new [SortedList<TKey,TValue>](#) is set to the number of elements in `dictionary`, so no resizing takes place while the list is being populated.

This constructor uses the default comparer for `TKey`. To specify a comparer, use the [SortedList<TKey,TValue>\(IDictionary<TKey,TValue>, IComparer<TKey>\)](#) constructor. The default comparer [Comparer<T>.Default](#) checks whether the key type `TKey` implements [System.IComparable<T>](#) and uses that implementation, if available. If not, [Comparer<T>.Default](#) checks whether the key type `TKey` implements [System.IComparable](#). If the key type `TKey` does not implement either interface, you can specify a [System.Collections.Generic.IComparer<T>](#) implementation in a constructor overload that accepts a `comparer` parameter.

The keys in `dictionary` are copied to the new [SortedList<TKey,TValue>](#) and sorted once, which makes this constructor an $O(n \log n)$ operation.

See also

- [IDictionary<TKey,TValue>](#)
- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedList<TKey,TValue>(Int32)

Initializes a new instance of the [SortedList<TKey,TValue>](#) class that is empty, has the specified initial capacity, and uses the default [IComparer<T>](#).

C#

```
public SortedList(int capacity);
```

Parameters

capacity Int32

The initial number of elements that the [SortedList<TKey,TValue>](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#)

`capacity` is less than zero.

Examples

The following code example creates a sorted list with an initial capacity of 4 and populates it with 4 entries.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string keys and
        // an initial capacity of 4.
        SortedList<string, string> openWith =
            new SortedList<string, string>(4);

        // Add 4 elements to the list.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // List the contents of the sorted list.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }
}
```

```
        kvp.Key, kvp.Value);
    }
}

/* This code example produces the following output:

Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/
```

Remarks

Every key in a [SortedList<TKey,TValue>](#) must be unique according to the default comparer.

The capacity of a [SortedList<TKey,TValue>](#) is the number of elements that the [SortedList<TKey,TValue>](#) can hold before resizing. As elements are added to a [SortedList<TKey,TValue>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [SortedList<TKey,TValue>](#).

The capacity can be decreased by calling [TrimExcess](#) or by setting the [Capacity](#) property explicitly. Decreasing the capacity reallocates memory and copies all the elements in the [SortedList<TKey,TValue>](#).

This constructor uses the default comparer for [TKey](#). To specify a comparer, use the [SortedList<TKey,TValue>\(Int32, IComparer<TKey>\)](#) constructor. The default comparer [Comparer<T>.Default](#) checks whether the key type [TKey](#) implements [System.IComparable<T>](#) and uses that implementation, if available. If not, [Comparer<T>.Default](#) checks whether the key type [TKey](#) implements [System.IComparable](#). If the key type [TKey](#) does not implement either interface, you can specify a [System.Collections.Generic.IComparer<T>](#) implementation in a constructor overload that accepts a [comparer](#) parameter.

This constructor is an O(n) operation, where n is [capacity](#).

See also

- [Capacity](#)

- Default
- [IComparable<T>](#)
- [IComparable](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedList<TKey,TValue> (IDictionary<TKey,TValue>, IComparer<TKey>)

Initializes a new instance of the [SortedList<TKey,TValue>](#) class that contains elements copied from the specified [IDictionary<TKey,TValue>](#), has sufficient capacity to accommodate the number of elements copied, and uses the specified [IComparer<T>](#).

C#

```
public SortedList(System.Collections.Generic.IDictionary<TKey, TValue>
dictionary, System.Collections.Generic.IComparer<TKey>? comparer);
```

Parameters

dictionary [IDictionary<TKey,TValue>](#)

The [IDictionary<TKey,TValue>](#) whose elements are copied to the new [SortedList<TKey,TValue>](#).

comparer [IComparer<TKey>](#)

The [IComparer<T>](#) implementation to use when comparing keys.

-or-

null to use the default [Comparer<T>](#) for the type of the key.

Exceptions

ArgumentNullException

`dictionary` is `null`.

ArgumentException

`dictionary` contains one or more duplicate keys.

Examples

The following code example shows how to use `SortedList<TKey,TValue>` to create a case-insensitive sorted copy of the information in a case-insensitive `Dictionary<TKey,TValue>`, by passing the `Dictionary<TKey,TValue>` to the `SortedList<TKey,TValue>(IDictionary<TKey,TValue>, IComparer<TKey>)` constructor. In this example, the case-insensitive comparers are for the current culture.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new Dictionary of strings, with string keys and
        // a case-insensitive equality comparer for the current
        // culture.
        Dictionary<string, string> openWith =
            new Dictionary<string, string>
                (StringComparer.CurrentCultureIgnoreCase);

        // Add some elements to the dictionary.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("Bmp", "paint.exe");
        openWith.Add("DIB", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Create a SortedList of strings with string keys and a
        // case-insensitive equality comparer for the current culture,
        // and initialize it with the contents of the Dictionary.
        SortedList<string, string> copy =
            new SortedList<string, string>(openWith,
                StringComparer.CurrentCultureIgnoreCase);

        // List the sorted contents of the copy.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in copy )
        {
            Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
```

```

                kvp.Value);
            }
        }

/* This code example produces the following output:

Key = Bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/

```

Remarks

Every key in a [SortedList<TKey,TValue>](#) must be unique according to the specified comparer; likewise, every key in the source `dictionary` must also be unique according to the specified comparer.

The capacity of the new [SortedList<TKey,TValue>](#) is set to the number of elements in `dictionary`, so no resizing takes place while the list is being populated.

The keys in `dictionary` are copied to the new [SortedList<TKey,TValue>](#) and sorted once, which makes this constructor an $O(n \log n)$ operation.

See also

- [IDictionary<TKey,TValue>](#)
- [IComparer<T>](#)
- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

SortedList<TKey, TValue>(Int32, IComparer<TKey>)

Initializes a new instance of the [SortedList<TKey,TValue>](#) class that is empty, has the specified initial capacity, and uses the specified [IComparer<T>](#).

C#

```
public SortedList(int capacity, System.Collections.Generic.IComparer<TKey>?
comparer);
```

Parameters

capacity Int32

The initial number of elements that the [SortedList<TKey,TValue>](#) can contain.

comparer IComparer<TKey>

The [IComparer<T>](#) implementation to use when comparing keys.

-or-

`null` to use the default [Comparer<T>](#) for the type of the key.

Exceptions

[ArgumentOutOfRangeException](#)

`capacity` is less than zero.

Examples

The following code example creates a sorted list with an initial capacity of 5 and a case-insensitive comparer for the current culture. The example adds four elements, some with lower-case keys and some with upper-case keys. The example then attempts to add an element with a key that differs from an existing key only by case, catches the resulting exception, and displays an error message. Finally, the example displays the elements in case-insensitive sort order.

C#

```

using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string keys, an
        // initial capacity of 5, and a case-insensitive comparer.
        SortedList<string, string> openWith =
            new SortedList<string, string>(5,
                StringComparer.CurrentCultureIgnoreCase);

        // Add 4 elements to the list.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("DIB", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // Try to add a fifth element with a key that is the same
        // except for case; this would be allowed with the default
        // comparer.
        try
        {
            openWith.Add("BMP", "paint.exe");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("\nBMP is already in the sorted list.");
        }

        // List the contents of the sorted list.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}", kvp.Key,
                kvp.Value);
        }
    }
}

/* This code example produces the following output:

BMP is already in the sorted list.

Key = bmp, Value = paint.exe
Key = DIB, Value = paint.exe
Key = rtf, Value = wordpad.exe
Key = txt, Value = notepad.exe
*/

```

Remarks

Every key in a [SortedList<TKey,TValue>](#) must be unique according to the specified comparer.

The capacity of a [SortedList<TKey,TValue>](#) is the number of elements that the [SortedList<TKey,TValue>](#) can hold before resizing. As elements are added to a [SortedList<TKey,TValue>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [SortedList<TKey,TValue>](#).

The capacity can be decreased by calling [TrimExcess](#) or by setting the [Capacity](#) property explicitly. Decreasing the capacity reallocates memory and copies all the elements in the [SortedList<TKey,TValue>](#).

This constructor is an $O(n)$ operation, where n is `capacity`.

See also

- [Capacity](#)
- [IComparer<T>](#)
- [Default](#)
- [IComparable<T>](#)
- [IComparable](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.Capacity Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the number of elements that the [SortedDictionary< TKey, TValue >](#) can contain.

C#

```
public int Capacity { get; set; }
```

Property Value

[Int32](#)

The number of elements that the [SortedDictionary< TKey, TValue >](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#)

[Capacity](#) is set to a value that is less than [Count](#).

[OutOfMemoryException](#)

There is not enough memory available on the system.

Remarks

[Capacity](#) is the number of elements that the [SortedDictionary< TKey, TValue >](#) can store. [Count](#) is the number of elements that are actually in the [SortedDictionary< TKey, TValue >](#).

[Capacity](#) is always greater than or equal to [Count](#). If [Count](#) exceeds [Capacity](#) while adding elements, the capacity is increased by automatically reallocating the internal array before copying the old elements and adding the new elements.

The capacity can be decreased by calling [TrimExcess](#) or by setting the [Capacity](#) property explicitly. When the value of [Capacity](#) is set explicitly, the internal array is also reallocated to accommodate the specified capacity.

Retrieving the value of this property is an O(1) operation; setting the property is an O(n) operation, where n is the new capacity.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Count](#)
- [TrimExcess\(\)](#)

SortedList<TKey, TValue>.Comparer Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the [IComparer<T>](#) for the sorted list.

C#

```
public System.Collections.Generic.IComparer<TKey> Comparer { get; }
```

Property Value

[IComparer<TKey>](#)

The [IComparable<T>](#) for the current [SortedList<TKey,TValue>](#).

Remarks

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedList< TKey, TValue >.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of key/value pairs contained in the [SortedList< TKey, TValue >](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of key/value pairs contained in the [SortedList< TKey, TValue >](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Remarks

[Capacity](#) is the number of elements that the [SortedList< TKey, TValue >](#) can store. [Count](#) is the number of elements that are actually in the [SortedList< TKey, TValue >](#).

[Capacity](#) is always greater than or equal to [Count](#). If [Count](#) exceeds [Capacity](#) while adding elements, the capacity is increased by automatically reallocating the internal array before copying the old elements and adding the new elements.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Capacity](#)

SortedDictionary<TKey, TValue>.Item[TKey] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the value associated with the specified key.

C#

```
public TValue this[TKey key] { get; set; }
```

Parameters

key TKey

The key whose value to get or set.

Property Value

TValue

The value associated with the specified key. If the specified key is not found, a get operation throws a [KeyNotFoundException](#) and a set operation creates a new element using the specified key.

Implements

[Item\[TKey\]](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[KeyNotFoundException](#)

The property is retrieved and `key` does not exist in the collection.

Examples

The following code 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 also 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 sorted list.

This code example is part of a larger example provided for the [SortedList<TKey,TValue>](#) class.

C#

```
// 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";
```

C#

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

C#

```
// When a program often has to try keys that turn out not to
// be in the list, 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.");
}
```

Remarks

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[key]`.

A key cannot be `null`, but a value can be, if the type of values in the list, `TValue`, is a reference type.

If the key is not found when a value is being retrieved, `KeyNotFoundException` is thrown. If the key is not found when a value is being set, the key and value are added.

You can also use the `Item[]` property to add new elements by setting the value of a key that does not exist in the `SortedList<TKey,TValue>`; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the `SortedList<TKey,TValue>`, setting the `Item[]` property overwrites the old value. In contrast, the `Add` method does not modify existing elements.

The C# language uses the `this` keyword to define the indexers instead of implementing the `Item[]` property. Visual Basic implements `Item[]` as a default property, which provides the same indexing functionality.

Retrieving the value of this property is an $O(\log n)$ operation, where n is `Count`. Setting the property is an $O(\log n)$ operation if the key is already in the `SortedList<TKey,TValue>`. If the key is not in the list, setting the property is an $O(n)$ operation for unsorted data, or $O(\log n)$ if the new element is added at the end of the list. If insertion causes a resize, the operation is $O(n)$.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Add\(TKey, TValue\)](#)

SortedList< TKey, TValue >.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the keys in the [SortedList< TKey, TValue >](#), in sorted order.

C#

```
public System.Collections.Generic.IList< TKey > Keys { get; }
```

Property Value

[IList< TKey >](#)

A [IList< T >](#) containing the keys in the [SortedList< TKey, TValue >](#).

Examples

The following code example shows how to enumerate the keys in the sorted list using the [Keys](#) property, and how to enumerate the keys and values in the sorted list.

The example also shows how to use the [Keys](#) property for efficient indexed retrieval of keys.

This code is part of a larger example that can be compiled and executed. See [SortedList< TKey, TValue >](#).

C#

```
// To get the keys alone, use the Keys property.  
IList< string > ilistKeys = openWith.Keys;  
  
// The elements of the list are strongly typed with the  
// type that was specified for the SortedList keys.  
Console.WriteLine();  
foreach( string s in ilistKeys )  
{  
    Console.WriteLine("Key = {0}", s);  
}  
  
// The Keys property is an efficient way to retrieve  
// keys by index.
```

```
Console.WriteLine("\nIndexed retrieval using the Keys " +
    "property: Keys[2] = {0}", openWith.Keys[2]);
```

C#

```
// When you use foreach to enumerate list 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);
}
```

Remarks

The order of the keys in the [IList<T>](#) is the same as the order in the [SortedList<TKey,TValue>](#).

The returned [IList<T>](#) is not a static copy; instead, the [IList<T>](#) refers back to the keys in the original [SortedList<TKey,TValue>](#). Therefore, changes to the [SortedList<TKey,TValue>](#) continue to be reflected in the [IList<T>](#).

The collection returned by the [Keys](#) property provides an efficient way to retrieve keys by index. It is not necessary to regenerate the list when the property is accessed, because the list is just a wrapper for the internal array of keys. The following code shows the use of the [Keys](#) property for indexed retrieval of keys from a sorted list of elements with string keys:

C#

```
string v = mySortedList.Values[3];
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IList<T>](#)
- [Values](#)

SortedList< TKey, TValue >.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a collection containing the values in the [SortedList< TKey, TValue >](#).

C#

```
public System.Collections.Generic.IList<TValue> Values { get; }
```

Property Value

[IList< TValue >](#)

A [IList< T >](#) containing the values in the [SortedList< TKey, TValue >](#).

Examples

This code example shows how to enumerate the values in the sorted list using the [Values](#) property, and how to enumerate the keys and values in the sorted list.

The example also shows how to use the [Values](#) property for efficient indexed retrieval of values.

This code example is part of a larger example provided for the [SortedList< TKey, TValue >](#) class.

C#

```
// To get the values alone, use the Values property.
IList<string> ilistValues = openWith.Values;

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

// The Values property is an efficient way to retrieve
// values by index.
Console.WriteLine("\nIndexed retrieval using the Values " +
    "property: Values[2] = {0}", openWith.Values[2]);
```

C#

```
// When you use foreach to enumerate list 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);
}
```

Remarks

The order of the values in the [IList<T>](#) is the same as the order in the [SortedList< TKey,TValue >](#).

The returned [IList<T>](#) is not a static copy; instead, the [IList<T>](#) refers back to the values in the original [SortedList< TKey,TValue >](#). Therefore, changes to the [SortedList< TKey,TValue >](#) continue to be reflected in the [IList<T>](#).

The collection returned by the [Values](#) property provides an efficient way to retrieve values by index. It is not necessary to regenerate the list when the property is accessed, because the list is just a wrapper for the internal array of values. The following code shows the use of the [Values](#) property for indexed retrieval of values from a sorted list of strings:

C#

```
string v = mySortedList.Values[3];
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IList<T>](#)
- [Keys](#)

SortedList<TKey,TValue>.Add(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an element with the specified key and value into the [SortedList<TKey,TValue>](#).

C#

```
public void Add(TKey key, TValue value);
```

Parameters

key TKey

The key of the element to add.

value TValue

The value of the element to add. The value can be `null` for reference types.

Implements

[Add\(TKey, TValue\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

An element with the same key already exists in the [SortedList<TKey,TValue>](#).

Examples

The following code example creates an empty [SortedList<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.

This code example is part of a larger example provided for the `SortedList<TKey,TValue>` class.

C#

```
// Create a new sorted list of strings, with string
// keys.
SortedList<string, string> openWith =
    new SortedList<string, string>();

// Add some elements to the list. 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 list.
try
{
    openWith.Add("txt", "winword.exe");
}
catch (ArgumentException)
{
    Console.WriteLine("An element with Key = \"txt\" already exists.");
}
```

Remarks

A key cannot be `null`, but a value can be, if the type of values in the sorted list, `TValue`, is a reference type.

You can also use the `Item[]` property to add new elements by setting the value of a key that does not exist in the `SortedList<TKey,TValue>`; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the `SortedList<TKey,TValue>`, setting the `Item[]` property overwrites the old value. In contrast, the `Add` method does not modify existing elements.

If `Count` already equals `Capacity`, the capacity of the `SortedList<TKey,TValue>` is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

This method is an $O(n)$ operation for unsorted data, where n is `Count`. It is an $O(\log n)$ operation if the new element is added at the end of the list. If insertion causes a resize, the operation is $O(n)$.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove\(TKey\)](#)
- [Item\[TKey\]](#)
- [Add\(TKey, TValue\)](#)

SortedList< TKey, TValue >.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements from the [SortedList< TKey, TValue >](#).

C#

```
public void Clear();
```

Implements

[Clear\(\)](#) , [Clear\(\)](#)

Remarks

[Count](#) is set to zero, and references to other objects from elements of the collection are also released.

[Capacity](#) remains unchanged. To reset the capacity of the [SortedList< TKey, TValue >](#), call [TrimExcess](#) or set the [Capacity](#) property directly. Trimming an empty [SortedList< TKey, TValue >](#) sets the capacity of the [SortedList< TKey, TValue >](#) to the default capacity.

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- TrimExcess()
- Remove(TKey)
- RemoveAt(Int32)

SortedDictionary<TKey, TValue>.ContainsKey(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [SortedDictionary<TKey, TValue>](#) contains a specific key.

C#

```
public bool ContainsKey(TKey key);
```

Parameters

key TKey

The key to locate in the [SortedDictionary<TKey, TValue>](#).

Returns

Boolean

`true` if the [SortedDictionary<TKey, TValue>](#) contains an element with the specified key; otherwise, `false`.

Implements

[ContainsKey\(TKey\)](#) , [ContainsKey\(TKey\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [ContainsKey](#) method to test whether a key exists prior to calling the [Add](#) method. It also shows how to use the [TryGetValue](#) method to retrieve values, which is an efficient way to retrieve values when a program frequently tries keys.

that are not in the sorted list. Finally, it shows the least efficient way to test whether keys exist, by using the [Item\[\]](#) property (the indexer in C#).

This code example is part of a larger example provided for the [SortedList<TKey,TValue>](#) class.

C#

```
// 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"]);
}
```

C#

```
// When a program often has to try keys that turn out not to
// be in the list, 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.");
}
```

C#

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

Remarks

This method is an $O(\log n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IndexOfKey\(TKey\)](#)
- [ContainsValue\(TValue\)](#)

SortedDictionary<TKey, TValue>.ContainsValue(TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [SortedDictionary<TKey, TValue>](#) contains a specific value.

C#

```
public bool ContainsValue(TValue value);
```

Parameters

value TValue

The value to locate in the [SortedDictionary<TKey, TValue>](#). The value can be `null` for reference types.

Returns

Boolean

`true` if the [SortedDictionary<TKey, TValue>](#) contains an element with the specified value; otherwise, `false`.

Remarks

This method determines equality using the default comparer [Comparer<T>.Default](#) for the value type `TValue`. [Comparer<T>.Default](#) checks whether the value type `TValue` implements [System.IComparable<T>](#) and uses that implementation, if available. If not, [Comparer<T>.Default](#) checks whether the value type `TValue` implements [System.IComparable](#). If the value type `TValue` does not implement either interface, this method uses [Object.Equals](#).

This method performs a linear search; therefore, the average execution time is proportional to [Count](#). That is, this method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IndexOfValue\(TValue\)](#)
- [ContainsKey\(TKey\)](#)

SortedDictionary<TKey, TValue>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [SortedDictionary<TKey, TValue>](#).

C#

```
public  
System.Collections.Generic.IEnumerator<System.Collections.Generic.KeyValuePair<TKey,  
TValue>> GetEnumerator();
```

Returns

[IEnumerator<KeyValuePair<TKey, TValue>>](#)

An [IEnumerator<T>](#) of type [KeyValuePair<TKey, TValue>](#) for the [SortedDictionary<TKey, TValue>](#).

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

The dictionary is maintained in a sorted order using an internal tree. Every new element is positioned at the correct sort position, and the tree is adjusted to maintain the sort order whenever an element is removed. While enumerating, the sort order is maintained.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first

element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

SortedDictionary<TKey, TValue>.IndexOfKey(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches for the specified key and returns the zero-based index within the entire [SortedDictionary<TKey, TValue>](#).

C#

```
public int IndexOfKey(TKey key);
```

Parameters

key TKey

The key to locate in the [SortedDictionary<TKey, TValue>](#).

Returns

[Int32](#)

The zero-based index of **key** within the entire [SortedDictionary<TKey, TValue>](#), if found; otherwise, -1.

Exceptions

[ArgumentNullException](#)

key is **null**.

Remarks

This method performs a binary search; therefore, this method is an $O(\log n)$ operation, where **n** is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [IndexOfValue\(TValue\)](#)

SortedDictionary< TKey, TValue >.Index OfValue(TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches for the specified value and returns the zero-based index of the first occurrence within the entire [SortedDictionary< TKey, TValue >](#).

C#

```
public int IndexOfValue(TValue value);
```

Parameters

value TValue

The value to locate in the [SortedDictionary< TKey, TValue >](#). The value can be `null` for reference types.

Returns

[Int32](#)

The zero-based index of the first occurrence of `value` within the entire [SortedDictionary< TKey, TValue >](#), if found; otherwise, -1.

Remarks

This method determines equality using the default comparer [Comparer< T >.Default](#) for the value type `TValue`. [Comparer< T >.Default](#) checks whether the value type `TValue` implements [System.IComparable< T >](#) and uses that implementation, if available. If not, [Comparer< T >.Default](#) checks whether the value type `TValue` implements [System.IComparable](#). If the value type `TValue` does not implement either interface, this method uses [Object.Equals](#).

This method performs a linear search; therefore, the average execution time is proportional to [Count](#). That is, this method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsValue\(TValue\)](#)
- [IndexOfKey\(TKey\)](#)

SortedList<TKey,TValue>.Remove(TKey) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the element with the specified key from the [SortedList<TKey,TValue>](#).

C#

```
public bool Remove(TKey key);
```

Parameters

key TKey

The key of the element to remove.

Returns

[Boolean](#)

`true` if the element is successfully removed; otherwise, `false`. This method also returns `false` if `key` was not found in the original [SortedList<TKey,TValue>](#).

Implements

[Remove\(TKey\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to remove a key/value pair from the sorted list using the [Remove](#) method.

This code example is part of a larger example provided for the `SortedList<TKey,TValue>` class.

C#

```
// 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.");  
}
```

Remarks

This method performs a binary search; however, the elements are moved up to fill in the open spot, so this method is an $O(n)$ operation, where n is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [RemoveAt\(Int32\)](#)
- [Clear\(\)](#)
- [Add\(TKey, TValue\)](#)

SortedDictionary< TKey, TValue >.RemoveAt(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the element at the specified index of the [SortedDictionary< TKey, TValue >](#).

C#

```
public void RemoveAt(int index);
```

Parameters

index [Int32](#)

The zero-based index of the element to remove.

Exceptions

[ArgumentOutOfRangeException](#)

index is less than zero.

-or-

index is equal to or greater than [Count](#).

Remarks

The elements are moved up to fill the open spot, so this method is an $O(n)$ operation, where **n** is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove\(TKey\)](#)
- [Clear\(\)](#)
- [Add\(TKey, TValue\)](#)

SortedDictionary< TKey, TValue >.TrimExcess Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the capacity to the actual number of elements in the [SortedDictionary< TKey, TValue >](#), if that number is less than 90 percent of current capacity.

C#

```
public void TrimExcess();
```

Remarks

This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large [SortedDictionary< TKey, TValue >](#) can be considerable, however, so the [TrimExcess](#) method does nothing if the list is at more than 90 percent of capacity. This avoids incurring a large reallocation cost for a relatively small gain.

This method is an $O(n)$ operation, where n is [Count](#).

To reset a [SortedDictionary< TKey, TValue >](#) to its initial state, call the [Clear](#) method before calling [TrimExcess](#) method. Trimming an empty [SortedDictionary< TKey, TValue >](#) sets the capacity of the [SortedDictionary< TKey, TValue >](#) to the default capacity.

The capacity can also be set using the [Capacity](#) property.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [Clear\(\)](#)
- [Capacity](#)
- [Count](#)

SortedList< TKey, TValue >.TryGetValue(TKey, TValue) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the value associated with the specified key.

C#

```
public bool TryGetValue(TKey key, out TValue value);
```

Parameters

key TKey

The key whose value to get.

value TValue

When this method returns, the value associated with the specified key, if the key is found; otherwise, the default value for the type of the **value** parameter. This parameter is passed uninitialized.

Returns

Boolean

true if the [SortedList< TKey, TValue >](#) contains an element with the specified key; otherwise, **false**.

Implements

[TryGetValue\(TKey, TValue\)](#) , [TryGetValue\(TKey, TValue\)](#)

Exceptions

[ArgumentNullException](#)

key is **null**.

Examples

The example shows how to use the [TryGetValue](#) method as a more efficient way to retrieve values in a program that frequently tries keys that are not in the sorted list. For contrast, the example also shows how the [Item\[\]](#) property (the indexer in C#) throws exceptions when attempting to retrieve nonexistent keys.

This code example is part of a larger example provided for the [SortedList<TKey,TValue>](#) class.

C#

```
// When a program often has to try keys that turn out not to
// be in the list, 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.");
}
```

C#

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

Remarks

This method combines the functionality of the [ContainsKey](#) method and the [Item\[\]](#) property.

If the key is not found, then the `value` parameter gets the appropriate default value for the value type `TValue`; for example, zero (0) for integer types, `false` for Boolean types, and `null` for reference types.

This method performs a binary search; therefore, this method is an $O(\log n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [Item\[TKey\]](#)
- [ContainsValue\(TValue\)](#)
- [IndexOfValue\(TValue\)](#)

SortedDictionary<TKey, TValue>.ICollection<KeyValuePair<TKey, TValue>>.Add Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds a key/value pair to the [ICollection<T>](#).

C#

```
void  
ICollection<KeyValuePair<TKey, TValue>>.Add(System.Collections.Generic.KeyValuePair  
<TKey, TValue> keyValuePair);
```

Parameters

keyValuePair [KeyValuePair<TKey, TValue>](#)

The [KeyValuePair<TKey, TValue>](#) to add to the [ICollection<T>](#).

Implements

[Add\(T\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey, TValue>.ICollection<KeyValuePair<TKey, TValue>>.Contains Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [ICollection<T>](#) contains a specific element.

C#

```
bool  
ICollection<KeyValuePair<TKey, TValue>>.Contains(System.Collections.Generic.KeyValue  
ePair<TKey, TValue> keyValuePair);
```

Parameters

keyValuePair [KeyValuePair<TKey, TValue>](#)

The [KeyValuePair<TKey, TValue>](#) to locate in the [ICollection<T>](#).

Returns

[Boolean](#)

`true` if `keyValuePair` is found in the [ICollection<T>](#); otherwise, `false`.

Implements

[Contains\(T\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey, TValue>.ICollection<KeyValuePair<TKey, TValue>>.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection<T>](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void  
ICollection<KeyValuePair<TKey, TValue>>.CopyTo(System.Collections.Generic.KeyValuePair<TKey, TValue>[] array, int arrayIndex);
```

Parameters

array [KeyValuePair<TKey, TValue>\[\]](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection<T>](#).
The [Array](#) must have zero-based indexing.

arrayIndex [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than zero.

[ArgumentException](#)

The number of elements in the source [ICollection<T>](#) is greater than the available space from [arrayIndex](#) to the end of the destination [array](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary<TKey, TValue>.ICollection<KeyValuePair<TKey, TValue>>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [ICollection<T>](#) is read-only.

C#

```
bool  
System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKe  
y, TValue>>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [ICollection<T>](#) is read-only; otherwise, `false`. In the default implementation of [SortedDictionary<TKey, TValue>](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedList<TKey, TValue>.ICollection<KeyValuePair<TKey, TValue>>.Remove Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the first occurrence of a specific key/value pair from the [ICollection<T>](#).

C#

```
bool  
ICollection<KeyValuePair<TKey, TValue>>.Remove(System.Collections.Generic.KeyValuePair<TKey, TValue> keyValuePair);
```

Parameters

keyValuePair [KeyValuePair<TKey, TValue>](#)

The [KeyValuePair<TKey, TValue>](#) to remove from the [ICollection<T>](#).

Returns

[Boolean](#)

`true` if `keyValuePair` was successfully removed from the [ICollection<T>](#); otherwise, `false`. This method also returns `false` if `keyValuePair` was not found in the original [ICollection<T>](#).

Implements

[Remove\(T\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

Sorted List<TKey, TValue>.IDictionary<TKey, TValue>.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection<T>](#) containing the keys of the [IDictionary<TKey, TValue>](#).

C#

```
System.Collections.Generic.ICollection<TKey>
System.Collections.Generic.IDictionary<TKey, TValue>.Keys { get; }
```

Property Value

[ICollection<TKey>](#)

An [ICollection<T>](#) containing the keys of the [IDictionary<TKey, TValue>](#).

Implements

[Keys](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Sorted List<TKey, TValue>.IDictionary<TKey, TValue>.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection<T>](#) containing the values in the [IDictionary<TKey, TValue>](#).

C#

```
System.Collections.Generic.ICollection<TValue>
System.Collections.Generic.IDictionary<TKey, TValue>.Values { get; }
```

Property Value

[ICollection<TValue>](#)

An object containing the values in the [IDictionary<TKey, TValue>](#).

Implements

[Values](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey, TValue>> IEnumerable<KeyValuePair<TKey, TValue>>.GetEnumerator();
```

Returns

[IEnumerator<KeyValuePair<TKey, TValue>>](#)

An [IEnumerator<T>](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedList<TKey,TValue>.IReadOnlyDictionary<TKey,TValue>.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an enumerable collection that contains the keys in the read-only dictionary.

C#

```
System.Collections.Generic.IEnumerable<TKey>
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>.Keys { get; }
```

Property Value

[IEnumerable<TKey>](#)

An enumerable collection that contains the keys in the read-only dictionary.

Implements

[Keys](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedList<TKey,TValue>.IReadOnlyDictionary<TKey,TValue>.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an enumerable collection that contains the values in the read-only dictionary.

C#

```
System.Collections.Generic.IEnumerable<TValue>
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>.Values { get; }
```

Property Value

[IEnumerable<TValue>](#)

An enumerable collection that contains the values in the read-only dictionary.

Implements

[Values](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedDictionary< TKey, TValue >.ICollection.CopyTo(Array, Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32 \)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `arrayIndex` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the non-generic implementations of [ICollection.CopyTo](#) throw [InvalidCastException](#), whereas the generic implementations throw [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [CopyTo\(Array, Int32\)](#)

SortedList< TKey, TValue >.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [SortedList< TKey, TValue >](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock this object before accessing the collection.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

SortedList< TKey, TValue >.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [SortedList< TKey, TValue >](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

The [SyncRoot](#) property returns an object that can be used to synchronize access to the [ICollection](#). Synchronization is effective only if all threads lock this object before accessing the collection. The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;  
lock (ic.SyncRoot) {
```

```
// Access the collection.  
}
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

SortedDictionary< TKey, TValue >.IDictionary.Add(Object, Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an element with the provided key and value to the [IDictionary](#).

C#

```
void IDictionary.Add(object key, object value);
```

Parameters

key [Object](#)

The [Object](#) to use as the key of the element to add.

value [Object](#)

The [Object](#) to use as the value of the element to add.

Implements

[Add\(Object, Object\)](#)

Exceptions

[ArgumentNullException](#)

key is `null`.

[ArgumentException](#)

key is of a type that is not assignable to the key type [TKey](#) of the [IDictionary](#).

-or-

value is of a type that is not assignable to the value type [TValue](#) of the [IDictionary](#).

-or-

An element with the same key already exists in the [IDictionary](#).

Examples

The following code example shows how to access the `SortedList<TKey,TValue>` class through the `System.Collections.IDictionary` interface. The code example creates an empty `SortedList<TKey,TValue>` of strings with string keys and uses the `IDictionary.Add` method to add some elements. The example demonstrates that the `IDictionary.Add` method throws an `ArgumentException` when attempting to add a duplicate key, or when a key or value of the wrong data type is supplied.

The code example demonstrates the use of several other members of the `System.Collections.IDictionary` interface.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string keys,
        // and access it using the IDictionary interface.
        //

        IDictionary openWith = new SortedList<string, string>();

        // Add some elements to the sorted list. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
        try
        {
            openWith.Add(42, new Example());
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("An exception was caught for " +
                "IDictionary.Add. Exception message:\n\t{0}\n",
                ex.Message);
        }

        // The Add method throws an exception if the new key is
        // already in the sorted list.
        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 returns null if the key is of the wrong data
    // type.
    Console.WriteLine("The indexer returns null"
        + " if the key is of the wrong type:");
    Console.WriteLine("For key = 2, value = {0}.",
        openWith[2]);

    // The indexer throws an exception when setting a value
    // if the key is of the wrong data type.
    try
    {
        openWith[2] = "This does not get added.";
    }
    catch (ArgumentException)
    {
        Console.WriteLine("A key of the wrong type was specified"
            + " when assigning to the indexer.");
    }

    // Unlike the default Item property on the SortedList class
    // itself, IDictionary.Item does not throw an exception
    // if the requested key is not in the sorted list.
    Console.WriteLine("For key = \"tif\", value = {0}.",
        openWith["tif"]);

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

```

```

// IDictionary.Contains returns false if the wrong data
// type is supplied.
Console.WriteLine("openWith.Contains(29.7) returns {0}",
    openWith.Contains(29.7));

// When you use foreach to enumerate sorted list elements
// with the IDictionary interface, the elements are retrieved
// as DictionaryEntry objects instead of 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 icoll = openWith.Values;

// The elements of the collection are strongly typed
// with the type that was specified for values,
// even though the ICollection interface is not strongly
// typed.
Console.WriteLine();
foreach( string s in icoll )
{
    Console.WriteLine("Value = {0}", s);
}

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

// The elements of the collection are strongly typed
// with the type that was specified for keys,
// even though the ICollection interface is not strongly
// typed.
Console.WriteLine();
foreach( string s in icoll )
{
    Console.WriteLine("Key = {0}", s);
}

// Use the Remove method to remove a key/value pair. No
// exception is thrown if the wrong data type is supplied.
Console.WriteLine("\nRemove(\"dib\")");
openWith.Remove("dib");

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

/* This code example produces the following output:

```

```

An exception was caught for IDictionary.Add. Exception message:
The value "42" is not of type "System.String" and cannot be used in this
generic collection.
Parameter name: key

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
The indexer returns null if the key is of the wrong type:
For key = 2, value = .
A key of the wrong type was specified when assigning to the indexer.
For key = "tif", value = .
Value added for key = "ht": hypertrm.exe
openWith.Contains(29.7) returns False

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("dib")
Key "dib" is not found.
*/

```

Remarks

You can also use the [Item\[\]](#) property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the dictionary, setting the [Item\[\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

This method is an $O(n)$ operation for unsorted data, where n is [Count](#). It is an $O(\log n)$ operation if the new element is added at the end of the list. If insertion causes a resize, the operation is $O(n)$.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Item\[Object\]](#)

SortedList< TKey, TValue >.IDictionary. Contains(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the [IDictionary](#) contains an element with the specified key.

C#

```
bool IDictionary.Contains(object key);
```

Parameters

key [Object](#)

The key to locate in the [IDictionary](#).

Returns

[Boolean](#)

`true` if the [IDictionary](#) contains an element with the key; otherwise, `false`.

Implements

[Contains\(Object\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [IDictionary.Contains](#) method of the [System.Collections.IDictionary](#) interface with a [SortedList< TKey, TValue >](#). The example demonstrates that the method returns `false` if a key of the wrong data type is supplied.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedList<string, string>();

        // Add some elements to the sorted list. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
```

C#

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

// IDictionary.Contains returns false if the wrong data
// type is supplied.
Console.WriteLine("openWith.Contains(29.7) returns {0}",
    openWith.Contains(29.7));
```

C#

```
}
```

Remarks

This method returns `false` if `key` is of a type that is not assignable to the key type `TKey` of the `SortedList<TKey,TValue>`.

This method is an $O(\log n)$ operation, where `n` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ContainsKey\(TKey\)](#)
- [ContainsValue\(TValue\)](#)

SortedList< TKey, TValue >.IDictionary.Get Enumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an [IDictionaryEnumerator](#) for the [IDictionary](#).

C#

```
System.Collections.IDictionaryEnumerator IDictionary.GetEnumerator();
```

Returns

[IDictionaryEnumerator](#)

An [IDictionaryEnumerator](#) for the [IDictionary](#).

Implements

[GetEnumerator\(\)](#)

Examples

The following code example shows how to enumerate the key/value pairs in the sorted list by using the `foreach` statement (`For Each` in Visual Basic), which hides the use of the enumerator. In particular, note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
{
```

```
public static void Main()
{
    // Create a new sorted list of strings, with string keys,
    // and access it using the IDictionary interface.
    //
    IDictionary openWith = new SortedList<string, string>();

    // Add some elements to the sorted list. There are no
    // duplicate keys, but some of the values are duplicates.
    // IDictionary.Add throws an exception if incorrect types
    // are supplied for key or value.
    openWith.Add("txt", "notepad.exe");
    openWith.Add("bmp", "paint.exe");
    openWith.Add("dib", "paint.exe");
    openWith.Add("rtf", "wordpad.exe");
}
```

C#

```
// When you use foreach to enumerate sorted list elements
// with the IDictionary interface, the elements are retrieved
// as DictionaryEntry objects instead of KeyValuePair objects.
Console.WriteLine();
foreach( DictionaryEntry de in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
        de.Key, de.Value);
}
```

C#

```
}
```

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. `Reset` also brings the enumerator back to this position. At this position, `Entry` is undefined. Therefore, you must call `MoveNext` to advance the enumerator to the first element of the collection before reading the value of `Entry`.

[Entry](#) returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Entry](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Entry](#) is undefined. To set [Entry](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetEnumerator\(\)](#)
- [IDictionaryEnumerator](#)

SortedList< TKey, TValue >.IDictionary.IsFixedSize Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [IDictionary](#) has a fixed size.

C#

```
bool System.Collections.IDictionary.IsFixedSize { get; }
```

Property Value

[Boolean](#)

`true` if the [IDictionary](#) has a fixed size; otherwise, `false`. In the default implementation of [SortedList< TKey, TValue >](#), this property always returns `false`.

Implements

[IsFixedSize](#)

Remarks

A collection with a fixed size does not allow the addition or removal of elements after the collection is created, but it allows the modification of existing elements.

A collection with a fixed size is simply a collection with a wrapper that prevents adding and removing elements; therefore, if changes are made to the underlying collection, including the addition or removal of elements, the fixed-size collection reflects those changes.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsFixedSize](#)
- [IsReadOnly](#)

SortedList<TKey,TValue>.IDictionary.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether the [IDictionary](#) is read-only.

C#

```
bool System.Collections.IDictionary.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the [IDictionary](#) is read-only; otherwise, `false`. In the default implementation of [SortedList<TKey,TValue>](#), this property always returns `false`.

Implements

[IsReadOnly](#)

Remarks

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsReadOnly](#)
- [IsFixedSize](#)

SortedDictionary<TKey, TValue>.IDictionary. Item[Object] Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets or sets the element with the specified key.

C#

```
object? System.Collections.IDictionary.Item[object key] { get; set; }
```

Parameters

key [Object](#)

The key of the element to get or set.

Property Value

[Object](#)

The element with the specified key, or `null` if `key` is not in the dictionary or `key` is of a type that is not assignable to the key type `TKey` of the [SortedDictionary<TKey, TValue>](#).

Implements

[Item\[Object\]](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#)

A value is being assigned, and `key` is of a type that is not assignable to the key type `TKey` of the [SortedDictionary<TKey, TValue>](#).

-or-

A value is being assigned and is of a type that isn't assignable to the value type `TValue` of the `SortedDictionary<TKey,TValue>`.

Examples

The following code example shows how to use the `IDictionary.Item[]` property (the indexer in C#) of the `System.Collections.IDictionary` interface with a `SortedDictionary<TKey,TValue>`, and ways the property differs from the `SortedDictionary<TKey,TValue>.Item[]` property.

The example shows that, like the `SortedDictionary<TKey,TValue>.Item[]` property, the `SortedDictionary<TKey,TValue>.IDictionary.Item[]` property can change the value associated with an existing key and can be used to add a new key/value pair if the specified key is not in the sorted list. The example also shows that unlike the `SortedDictionary<TKey,TValue>.Item[]` property, the `SortedDictionary<TKey,TValue>.IDictionary.Item[]` property does not throw an exception if `key` is not in the sorted list, returning a null reference instead. Finally, the example demonstrates that getting the `SortedDictionary<TKey,TValue>.IDictionary.Item[]` property returns a null reference if `key` is not the correct data type, and that setting the property throws an exception if `key` is not the correct data type.

The code example is part of a larger example, including output, provided for the `IDictionary.Add` method.

C#

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

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedDictionary<string, string>();

        // Add some elements to the sorted list. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
```

C#

```
// 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 returns null if the key is of the wrong data
// type.
Console.WriteLine("The indexer returns null"
    + " if the key is of the wrong type:");
Console.WriteLine("For key = 2, value = {0}.",
    openWith[2]);

// The indexer throws an exception when setting a value
// if the key is of the wrong data type.
try
{
    openWith[2] = "This does not get added.";
}
catch (ArgumentException)
{
    Console.WriteLine("A key of the wrong type was specified"
        + " when assigning to the indexer.");
}
```

C#

```
// Unlike the default Item property on the SortedList class
// itself, IDictionary.Item does not throw an exception
// if the requested key is not in the sorted list.
Console.WriteLine("For key = \"tif\", value = {0}.",
    openWith["tif"]);
```

C#

```
}
```

Remarks

This property returns `null` if `key` is of a type that is not assignable to the key type `TKey` of the `SortedList<TKey,TValue>`.

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[key]`.

You can also use the `Item[]` property to add new elements by setting the value of a key that does not exist in the dictionary; for example, `myCollection["myNonexistentKey"] = myValue`.

However, if the specified key already exists in the dictionary, setting the `Item[]` property overwrites the old value. In contrast, the `Add` method does not modify existing elements.

The C# language uses the `this` keyword to define the indexers instead of implementing the `IDictionary.Item[]` property. Visual Basic implements `IDictionary.Item[]` as a default property, which provides the same indexing functionality.

Retrieving the value of this property is an $O(\log n)$ operation, where n is `Count`. Setting the property is an $O(\log n)$ operation if the key is already in the `SortedList<TKey,TValue>`. If the key is not in the list, setting the property is an $O(n)$ operation for unsorted data, or $O(\log n)$ if the new element is added at the end of the list. If insertion causes a resize, the operation is $O(n)$.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Item\[TKey\]](#)
- [Add\(Object, Object\)](#)

SortedDictionary< TKey, TValue >.IDictionary.Keys Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection](#) containing the keys of the [IDictionary](#).

C#

```
System.Collections.ICollection System.Collections.IDictionary.Keys { get; }
```

Property Value

[ICollection](#)

An [ICollection](#) containing the keys of the [IDictionary](#).

Implements

[Keys](#)

Examples

The following code example shows how to use the [Keys](#) property of the [System.Collections.IDictionary](#) interface with a [SortedDictionary< TKey, TValue >](#), to list the keys in the dictionary. The example also shows how to enumerate the key/value pairs in the sorted list; note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
```

```
{  
    public static void Main()  
    {  
        // Create a new sorted list of strings, with string keys,  
        // and access it using the IDictionary interface.  
        //  
        IDictionary openWith = new SortedList<string, string>();  
  
        // Add some elements to the sorted list. There are no  
        // duplicate keys, but some of the values are duplicates.  
        // IDictionary.Add throws an exception if incorrect types  
        // are supplied for key or value.  
        openWith.Add("txt", "notepad.exe");  
        openWith.Add("bmp", "paint.exe");  
        openWith.Add("dib", "paint.exe");  
        openWith.Add("rtf", "wordpad.exe");
```

C#

```
// To get the keys alone, use the Keys property.  
icoll = openWith.Keys;  
  
// The elements of the collection are strongly typed  
// with the type that was specified for keys,  
// even though the ICollection interface is not strongly  
// typed.  
Console.WriteLine();  
foreach( string s in icoll )  
{  
    Console.WriteLine("Key = {0}", s);  
}
```

C#

```
// When you use foreach to enumerate sorted list elements  
// with the IDictionary interface, the elements are retrieved  
// as DictionaryEntry objects instead of KeyValuePair objects.  
Console.WriteLine();  
foreach( DictionaryEntry de in openWith )  
{  
    Console.WriteLine("Key = {0}, Value = {1}",  
        de.Key, de.Value);  
}
```

C#

```
}  
}
```

Remarks

The order of the keys in the [ICollection](#) is the same as the order in the [SortedList<TKey,TValue>](#).

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)
- [Keys](#)
- [Values](#)

SortedList< TKey, TValue >.IDictionary.Remove(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes the element with the specified key from the [IDictionary](#).

C#

```
void IDictionary.Remove(object key);
```

Parameters

key [Object](#)

The key of the element to remove.

Implements

[Remove\(Object\)](#)

Exceptions

[ArgumentNullException](#)

`key` is `null`.

Examples

The following code example shows how to use the [IDictionary.Remove](#) of the [System.Collections.IDictionary](#) interface with a [SortedList< TKey, TValue >](#).

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

```

public class Example
{
    public static void Main()
    {
        // Create a new sorted list of strings, with string keys,
        // and access it using the IDictionary interface.
        //
        IDictionary openWith = new SortedList<string, string>();

        // Add some elements to the sorted list. There are no
        // duplicate keys, but some of the values are duplicates.
        // IDictionary.Add throws an exception if incorrect types
        // are supplied for key or value.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
    }
}

```

C#

```

// Use the Remove method to remove a key/value pair. No
// exception is thrown if the wrong data type is supplied.
Console.WriteLine("\nRemove(\"dib\")");
openWith.Remove("dib");

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

```

C#

```

}
}
```

Remarks

This method performs a binary search; however, the elements are moved up to fill in the open spot, so this method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Remove\(TKey\)](#)
- [RemoveAt\(Int32\)](#)

SortedList< TKey, TValue >.IDictionary.Values Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an [ICollection](#) containing the values in the [IDictionary](#).

C#

```
System.Collections.ICollection System.Collections.IDictionary.Values { get; }
```

Property Value

[ICollection](#)

An [ICollection](#) containing the values in the [IDictionary](#).

Implements

[Values](#)

Examples

The following code example shows how to use the [IDictionary.Values](#) property of the [System.Collections.IDictionary](#) interface with a [SortedList< TKey, TValue >](#), to list the values in the sorted list. The example also shows how to enumerate the key/value pairs in the sorted list; note that the enumerator for the [System.Collections.IDictionary](#) interface returns [DictionaryEntry](#) objects rather than [KeyValuePair< TKey, TValue >](#) objects.

The code example is part of a larger example, including output, provided for the [IDictionary.Add](#) method.

C#

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

public class Example
```

```
{  
    public static void Main()  
    {  
        // Create a new sorted list of strings, with string keys,  
        // and access it using the IDictionary interface.  
        //  
        IDictionary openWith = new SortedList<string, string>();  
  
        // Add some elements to the sorted list. There are no  
        // duplicate keys, but some of the values are duplicates.  
        // IDictionary.Add throws an exception if incorrect types  
        // are supplied for key or value.  
        openWith.Add("txt", "notepad.exe");  
        openWith.Add("bmp", "paint.exe");  
        openWith.Add("dib", "paint.exe");  
        openWith.Add("rtf", "wordpad.exe");
```

C#

```
// To get the values alone, use the Values property.  
ICollection icoll = openWith.Values;  
  
// The elements of the collection are strongly typed  
// with the type that was specified for values,  
// even though the ICollection interface is not strongly  
// typed.  
Console.WriteLine();  
foreach( string s in icoll )  
{  
    Console.WriteLine("Value = {0}", s);  
}
```

C#

```
// When you use foreach to enumerate sorted list elements  
// with the IDictionary interface, the elements are retrieved  
// as DictionaryEntry objects instead of KeyValuePair objects.  
Console.WriteLine();  
foreach( DictionaryEntry de in openWith )  
{  
    Console.WriteLine("Key = {0}, Value = {1}",  
        de.Key, de.Value);  
}
```

C#

```
}  
}
```

Remarks

The order of the values in the [ICollection](#) is the same as the order in the [SortedList<TKey,TValue>](#).

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ICollection](#)
- [Keys](#)
- [Values](#)

SortedDictionary<TKey, TValue>.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetEnumerator\(\)](#)
- [IEnumerator](#)

SortedSet<T>.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [SortedSet<T>](#) object.

C#

```
public struct SortedSet<T>.Enumerator : System.Collections.Generic.IEnumerator<T>,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

Type Parameters

T

Inheritance [Object](#) → [ValueType](#) → [SortedSet<T>.Enumerator](#)

Implements [IEnumerator<T>](#) , [IEnumerator](#) , [IDisposable](#) , [IDeserializationCallback](#) ,
[ISerializable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, the [Current](#) property is undefined. Therefore, you must call the [MoveNext](#) method to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator object instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator doesn't have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in the [System.Collections.Generic](#) namespace are not synchronized.

Properties

[+] [Expand table](#)

Current	Gets the element at the current position of the enumerator.
-------------------------	---

Methods

[+] [Expand table](#)

Dispose()	Releases all resources used by the SortedSet<T>.Enumerator .
---------------------------	--

MoveNext()	Advances the enumerator to the next element of the SortedSet<T> collection.
----------------------------	---

Explicit Interface Implementations

[+] [Expand table](#)

IDeserializationCallback.OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
--	---

IEnumerator.Current	Gets the element at the current position of the enumerator.
IEnumerator.Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.
ISerializable.GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data needed to serialize the <code>SortedSet<T></code> instance.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.
- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [SortedSet<T>.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [SortedSet<T>](#) collection.

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to the [MoveNext](#) method advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[IEnumerator.Current](#) does not move the position of the enumerator, and consecutive calls to [IEnumerator.Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Enumerator.IDeserializationCallback.OnDeserialization Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and raises the deserialization event when the deserialization is complete.

C#

```
void IDeserializationCallback.OnDeserialization(object sender);
```

Parameters

sender [Object](#)

The source of the deserialization event.

Implements

[OnDeserialization\(Object\)](#)

Exceptions

[SerializationException](#)

The [SerializationInfo](#) object associated with the current [SortedSet<T>](#) instance is invalid.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

SortedSet<T>.Enumerator.ISerializable.GetObjectData Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and returns the data needed to serialize the [SortedSet<T>](#) instance.

C#

```
void ISerializable.GetObjectData(System.Runtime.Serialization.SerializationInfo  
info, System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information required to serialize the [SortedSet<T>](#) instance.

context [StreamingContext](#)

A [StreamingContext](#) object that contains the source and destination of the serialized stream associated with the [SortedSet<T>](#) instance.

Implements

[GetObjectData\(SerializationInfo, StreamingContext\)](#)

Exceptions

[ArgumentNullException](#)

`info` is `null`.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

SortedSet<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a collection of objects that is maintained in sorted order.

C#

```
public class SortedSet<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>,
System.Collections.Generic.IReadOnlyCollection<T>,
System.Collections.Generic.IReadOnlySet<T>, System.Collections.Generic.ISet<T>,
System.Collections.ICollection,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

Type Parameters

T

The type of elements in the set.

Inheritance [Object](#) → [SortedSet<T>](#)

Implements [ICollection<T>](#) , [IEnumerable<T>](#) , [IReadOnlyCollection<T>](#) , [ISet<T>](#) ,
[ICollection](#) , [IEnumerable](#) , [IReadOnlySet<T>](#) , [IDeserializationCallback](#) ,
[ISerializable](#)

Examples

The following example demonstrates a [SortedSet<T>](#) class that is created with the constructor that takes an [IComparer<T>](#) as a parameter. This comparer ([ByFileExtension](#)) is used to sort a list of file names by their extensions.

This example demonstrates how to create a sorted set of media file names, remove unwanted elements, view a range of elements, and compare the set with another sorted set.

C#

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

class Program
{
    static void Main(string[] args)
    {
        try
        {
            // Get a list of the files to use for the sorted set.
            IEnumerable<string> files1 =
                Directory.EnumerateFiles(@"\\archives\2007\media",
                "*", SearchOption.AllDirectories);

            // Create a sorted set using the ByFileExtension comparer.
            var mediaFiles1 = new SortedSet<string>(new ByFileExtension());

            // Note that there is a SortedSet constructor that takes an
            IEnumerable,
            // but to remove the path information they must be added individually.
            foreach (string f in files1)
            {
                mediaFiles1.Add(f.Substring(f.LastIndexOf("\\") + 1));
            }

            // Remove elements that have non-media extensions.
            // See the 'IsDoc' method.
            Console.WriteLine("Remove docs from the set...");
            Console.WriteLine($"\\tCount before: {mediaFiles1.Count}");
            mediaFiles1.RemoveWhere(IsDoc);
            Console.WriteLine($"\\tCount after: {mediaFiles1.Count}");

            Console.WriteLine();

            // List all the avi files.
            SortedSet<string> aviFiles = mediaFiles1.GetViewBetween("avi", "avj");

            Console.WriteLine("AVI files:");
            foreach (string avi in aviFiles)
            {
                Console.WriteLine($"\\t{avi}");
            }

            Console.WriteLine();

            // Create another sorted set.
            IEnumerable<string> files2 =
                Directory.EnumerateFiles(@"\\archives\2008\media",
                "*", SearchOption.AllDirectories);

            var mediaFiles2 = new SortedSet<string>(new ByFileExtension());
```

```

        foreach (string f in files2)
    {
        mediaFiles2.Add(f.Substring(f.LastIndexOf(@"\") + 1));
    }

    // Remove elements in mediaFiles1 that are also in mediaFiles2.
    Console.WriteLine("Remove duplicates (of mediaFiles2) from the
set...");

    Console.WriteLine($"\\tCount before: {mediaFiles1.Count}");
    mediaFiles1.ExceptWith(mediaFiles2);
    Console.WriteLine($"\\tCount after: {mediaFiles1.Count}");

    Console.WriteLine();

    Console.WriteLine("List of mediaFiles1:");
    foreach (string f in mediaFiles1)
    {
        Console.WriteLine($"\\t{f}");
    }

    // Create a set of the sets.
    IEqualityComparer<SortedSet<string>> comparer =
        SortedSet<string>.CreateSetComparer();

    var allMedia = new HashSet<SortedSet<string>>(comparer);
    allMedia.Add(mediaFiles1);
    allMedia.Add(mediaFiles2);
}

catch(IOException ioEx)
{
    Console.WriteLine(ioEx.Message);
}

catch (UnauthorizedAccessException AuthEx)
{
    Console.WriteLine(AuthEx.Message);
}
}

// Defines a predicate delegate to use
// for the SortedSet.RemoveWhere method.
private static bool IsDoc(string s)
{
    s = s.ToLower();
    return (s.EndsWith(".txt") ||
        s.EndsWith(".xls") ||
        s.EndsWith(".xlsx") ||
        s.EndsWith(".pdf") ||
        s.EndsWith(".doc") ||
        s.EndsWith(".docx"));
}

// Defines a comparer to create a sorted set
// that is sorted by the file extensions.

```

```

public class ByFileExtension : IComparer<string>
{
    string xExt, yExt;

    CaseInsensitiveComparer caseiComp = new CaseInsensitiveComparer();

    public int Compare(string x, string y)
    {
        // Parse the extension from the file name.
        xExt = x.Substring(x.LastIndexOf(".") + 1);
        yExt = y.Substring(y.LastIndexOf(".") + 1);

        // Compare the file extensions.
        int vExt = caseiComp.Compare(xExt, yExt);
        if (vExt != 0)
        {
            return vExt;
        }
        else
        {
            // The extension is the same,
            // so compare the filenames.
            return caseiComp.Compare(x, y);
        }
    }
}

```

Remarks

A `SortedSet<T>` object maintains a sorted order without affecting performance as elements are inserted and deleted. Duplicate elements are not allowed. Changing the sort values of existing items is not supported and may lead to unexpected behavior.

For a thread safe alternative to `SortedSet<T>`, see [ImmutableSortedSet<T>](#)

Constructors

[] [Expand table](#)

<code>SortedSet<T>()</code>	Initializes a new instance of the <code>SortedSet<T></code> class.
<code>SortedSet<T>(IComparer<T>)</code>	Initializes a new instance of the <code>SortedSet<T></code> class that uses a specified comparer.
<code>SortedSet<T>(IEnumerable<T>, IComparer<T>)</code>	Initializes a new instance of the <code>SortedSet<T></code> class that contains elements copied from a specified enumerable collection and that uses a specified comparer.

<code>SortedSet<T>(IEnumerable<T>)</code>	Initializes a new instance of the <code>SortedSet<T></code> class that contains elements copied from a specified enumerable collection.
<code>SortedSet<T>(SerializationInfo, StreamingContext)</code>	Initializes a new instance of the <code>SortedSet<T></code> class that contains serialized data.

Properties

 [Expand table](#)

<code>Comparer</code>	Gets the <code>IComparer<T></code> object that is used to order the values in the <code>SortedSet<T></code> .
<code>Count</code>	Gets the number of elements in the <code>SortedSet<T></code> .
<code>Max</code>	Gets the maximum value in the <code>SortedSet<T></code> , as defined by the comparer.
<code>Min</code>	Gets the minimum value in the <code>SortedSet<T></code> , as defined by the comparer.

Methods

 [Expand table](#)

<code>Add(T)</code>	Adds an element to the set and returns a value that indicates if it was successfully added.
<code>Clear()</code>	Removes all elements from the set.
<code>Contains(T)</code>	Determines whether the set contains a specific element.
<code>CopyTo(T[], Int32, Int32)</code>	Copies a specified number of elements from <code>SortedSet<T></code> to a compatible one-dimensional array, starting at the specified array index.
<code>CopyTo(T[], Int32)</code>	Copies the complete <code>SortedSet<T></code> to a compatible one-dimensional array, starting at the specified array index.
<code>CopyTo(T[])</code>	Copies the complete <code>SortedSet<T></code> to a compatible one-dimensional array, starting at the beginning of the target array.
<code>CreateSetComparer()</code>	Returns an <code>IEqualityComparer</code> object that can be used to create a collection that contains individual sets.
<code>CreateSetComparer(IEqualityComparer<T>)</code>	Returns an <code>IEqualityComparer</code> object, according to a specified comparer, that can be used to create a collection that contains individual sets.

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
ExceptWith(IEnumerable<T>)	Removes all elements that are in a specified collection from the current SortedSet<T> object.
GetEnumerator()	Returns an enumerator that iterates through the SortedSet<T> .
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface and returns the data that you must have to serialize a SortedSet<T> object.
GetType()	Gets the Type of the current instance. (Inherited from Object)
GetViewBetween(T, T)	Returns a view of a subset in a SortedSet<T> .
IntersectWith(IEnumerable<T>)	Modifies the current SortedSet<T> object so that it contains only elements that are also in a specified collection.
IsProperSubsetOf(IEnumerable<T>)	Determines whether a SortedSet<T> object is a proper subset of the specified collection.
IsProperSupersetOf(IEnumerable<T>)	Determines whether a SortedSet<T> object is a proper superset of the specified collection.
IsSubsetOf(IEnumerable<T>)	Determines whether a SortedSet<T> object is a subset of the specified collection.
IsSupersetOf(IEnumerable<T>)	Determines whether a SortedSet<T> object is a superset of the specified collection.
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 completed.
Overlaps(IEnumerable<T>)	Determines whether the current SortedSet<T> object and a specified collection share common elements.
Remove(T)	Removes a specified item from the SortedSet<T> .
RemoveWhere(Predicate<T>)	Removes all elements that match the conditions defined by the specified predicate from a SortedSet<T> .
Reverse()	Returns an IEnumerable<T> that iterates over the SortedSet<T> in reverse order.

SetEquals(IEnumerable<T>)	Determines whether the current SortedSet<T> object and the specified collection contain the same elements.
SymmetricExceptWith(IEnumerable<T>)	Modifies the current SortedSet<T> object so that it contains only elements that are present either in the current object or in the specified collection, but not both.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TryGetValue(T, T)	Searches the set for a given value and returns the equal value it finds, if any.
UnionWith(IEnumerable<T>)	Modifies the current SortedSet<T> object so that it contains all elements that are present in either the current object or the specified collection.

Explicit Interface Implementations

 [Expand table](#)

ICollection.CopyTo(Array, Int32)	Copies the complete SortedSet<T> to a compatible one-dimensional array, starting at the specified array index.
ICollection.IsSynchronized	Gets a value that indicates whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
ICollection<T>.Add(T)	Adds an item to an ICollection<T> object.
ICollection<T>.IsReadOnly	Gets a value that indicates whether a ICollection is read-only.
IDeserializationCallback.OnDeserialization(Object)	Implements the IDeserializationCallback interface, and raises the deserialization event when the deserialization is completed.
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through a collection.
IEnumerable<T>.GetEnumerator()	Returns an enumerator that iterates through a collection.
ISerializable.GetObjectData(SerializationInfo, StreamingContext)	Implements the ISerializable interface, and returns the data that you need to serialize the SortedSet<T> instance.

Extension Methods

<code>ToImmutableArray<TSource>(IEnumerable<TSource>)</code>	Creates an immutable array from the specified collection.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Constructs an immutable dictionary based on some transformation of a sequence.
<code>ToImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
<code>ToImmutableDictionary<TSource,TKey TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>, IEqualityComparer<TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.

<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer<TKey>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set of its contents.
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillEventHandler)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)</code>	Copies <code>DataRow</code> objects to the specified <code>DataTable</code> , given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>CopyToDataTable<T>(IEnumerable<T>)</code>	Returns a <code>DataTable</code> that contains copies of the <code>DataRow</code> objects, given an input <code>IEnumerable<T></code> object where the generic parameter <code>T</code> is <code>DataRow</code> .
<code>Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)</code>	Applies an accumulator function over a sequence.
<code>Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
<code>Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)</code>	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.

All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether all elements of a sequence satisfy a condition.
Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether any element of a sequence satisfies a condition.
Any<TSource>(IEnumerable<TSource>)	Determines whether a sequence contains any elements.
Append<TSource>(IEnumerable<TSource>, TSource)	Appends a value to the end of the sequence.
AsEnumerable<TSource>(IEnumerable<TSource>)	Returns the input typed as IEnumerable<T> .
Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the average of a sequence of Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the average of a sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the average of a sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on

	each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in

	a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.
Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.

<code>Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set difference of two sequences by using the default equality comparer to compare values.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set difference of two sequences according to a specified key selector function.
<code>First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element in a sequence that satisfies a specified condition.
<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key

<code>Comparer<TKey></code>	selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The keys are compared by using a specified comparer.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on key equality and groups the results. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>,</code>	Correlates the elements of two sequences based on equality of

<code>Func<TOuter, IEnumerable<TInner>, TResult>()</code>	keys and groups the results. The default equality comparer is used to compare keys.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Produces the set intersection of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
<code>Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)</code>	Produces the set intersection of two sequences by using the default equality comparer to compare values.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>, IEqualityComparer<TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>IntersectBy<TSource, TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource, TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter, TInner, TKey, TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter, TKey>, Func<TInner, TKey>, Func<TOuter, TInner, TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.

<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an Int64 that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an Int64 that represents the total number of elements in a sequence.
<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.
<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int32 value.

<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Decimal value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Double value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int32 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the minimum Single value.
<code>Min<TSource>(IEnumerable<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
<code>MinBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the minimum value in a generic sequence according to a specified key selector function.

<code>OfType<TResult>(IEnumerable)</code>	Filters the elements of an IEnumerable based on a specified type.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in ascending order by using a specified comparer.
<code>OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in ascending order according to a key.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Sorts the elements of a sequence in descending order by using a specified comparer.
<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an IEnumerable<T> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an IEnumerable<T> , flattens the resulting sequences into one sequence, and invokes a

	<p>result selector function on each element therein.</p>
SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)	<p>Projects each element of a sequence to an IEnumerable<T>, flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	<p>Determines whether two sequences are equal by comparing their elements by using a specified IEqualityComparer<T>.</p>
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	<p>Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.</p>
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.</p>
Single<TSource>(IEnumerable<TSource>)	<p>Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	<p>Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	<p>Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.</p>
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	<p>Returns the only element of a sequence that satisfies a specified</p>

	condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform

	function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Take<TSource>(IEnumerable<TSource>, Int32)	Returns a specified number of contiguous elements from the start of a sequence.

<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function.
<code>ToArray<TSource>(IEnumerable<TSource>)</code>	Creates an array from a <code>IEnumerable<T></code> .
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function and key comparer.
<code>ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to a specified key selector function, a comparer, and an element selector function.
<code>ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Creates a <code>Dictionary<TKey, TValue></code> from an <code>IEnumerable<T></code> according to specified key selector and element selector functions.
<code>ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> using the <code>comparer</code> to compare keys.
<code>ToHashSet<TSource>(IEnumerable<TSource>)</code>	Creates a <code>HashSet<T></code> from an <code>IEnumerable<T></code> .
<code>ToList<TSource>(IEnumerable<TSource>)</code>	Creates a <code>List<T></code> from an <code>IEnumerable<T></code> .

ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

<code>Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)</code>	Produces a sequence of tuples with elements from the two specified sequences.
<code>Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)</code>	Produces a sequence of tuples with elements from the three specified sequences.
<code>Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)</code>	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.
<code>AsParallel(IEnumerable)</code>	Enables parallelization of a query.
<code>AsParallel<TSource>(IEnumerable<TSource>)</code>	Enables parallelization of a query.
<code>AsQueryable(IEnumerable)</code>	Converts an IEnumerable to an IQueryable .
<code>AsQueryable<TElement>(IEnumerable<TElement>)</code>	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
<code>Ancestors<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
<code>Ancestors<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the ancestors of every node in the source collection.
<code>DescendantNodes<T>(IEnumerable<T>)</code>	Returns a collection of the descendant nodes of every document and element in the source collection.
<code>Descendants<T>(IEnumerable<T>, XName)</code>	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
<code>Descendants<T>(IEnumerable<T>)</code>	Returns a collection of elements that contains the descendant

	elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.
InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ImmutableSortedSet<T>](#)

SortedSet<T> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [SortedSet<T>](#) class.

Overloads

[+] [Expand table](#)

SortedSet<T>()	Initializes a new instance of the SortedSet<T> class.
SortedSet<T>(IComparer<T>)	Initializes a new instance of the SortedSet<T> class that uses a specified comparer.
SortedSet<T>(IEnumerable<T>)	Initializes a new instance of the SortedSet<T> class that contains elements copied from a specified enumerable collection.
SortedSet<T>(IEnumerable<T>, IComparer<T>)	Initializes a new instance of the SortedSet<T> class that contains elements copied from a specified enumerable collection and that uses a specified comparer.
SortedSet<T>(SerializationInfo, StreamingContext)	Initializes a new instance of the SortedSet<T> class that contains serialized data.

Remarks

This constructor is an $O(1)$ operation.

SortedSet<T>()

Initializes a new instance of the [SortedSet<T>](#) class.

C#

```
public SortedSet();
```

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>(IComparer<T>)

Initializes a new instance of the [SortedSet<T>](#) class that uses a specified comparer.

C#

```
public SortedSet(System.Collections.Generic.IComparer<T>? comparer);
```

Parameters

comparer [IComparer<T>](#)

The default comparer to use for comparing objects.

Exceptions

[ArgumentNullException](#)

`comparer` is `null`.

Examples

The following example defines a comparer (`ByFileExtension`) that is used to construct a sorted set that sorts file names by their extensions. This code example is part of a larger example provided for the [SortedSet<T>](#) class.

C#

```
// Create a sorted set using the ByFileExtension comparer.  
var mediaFiles1 = new SortedSet<string>(new ByFileExtension());
```

C#

```
// Defines a comparer to create a sorted set
// that is sorted by the file extensions.
public class ByFileExtension : IComparer<string>
{
    string xExt, yExt;

    CaseInsensitiveComparer caseiComp = new CaseInsensitiveComparer();

    public int Compare(string x, string y)
    {
        // Parse the extension from the file name.
        xExt = x.Substring(x.LastIndexOf(".") + 1);
        yExt = y.Substring(y.LastIndexOf(".") + 1);

        // Compare the file extensions.
        int vExt = caseiComp.Compare(xExt, yExt);
        if (vExt != 0)
        {
            return vExt;
        }
        else
        {
            // The extension is the same,
            // so compare the filenames.
            return caseiComp.Compare(x, y);
        }
    }
}
```

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>(IEnumerable<T>)

Initializes a new instance of the [SortedSet<T>](#) class that contains elements copied from a specified enumerable collection.

C#

```
public SortedSet(System.Collections.Generic.IEnumerable<T> collection);
```

Parameters

collection [IEnumerable<T>](#)

The enumerable collection to be copied.

Remarks

Duplicate elements in the enumerable collection are not copied into the new instance of the [SortedSet<T>](#) class, and no exceptions are thrown.

This constructor is an $O(n \log n)$ operation, where n is the number of elements in the `collection` parameter.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>(IEnumerable<T>, IComparer<T>)

Initializes a new instance of the [SortedSet<T>](#) class that contains elements copied from a specified enumerable collection and that uses a specified comparer.

C#

```
public SortedSet(System.Collections.Generic.IEnumerable<T> collection,
```

```
System.Collections.Generic.IComparer<T> comparer);
```

Parameters

collection [IEnumerable<T>](#)

The enumerable collection to be copied.

comparer [IComparer<T>](#)

The default comparer to use for comparing objects.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>(SerializationInfo, StreamingContext)

Initializes a new instance of the [SortedSet<T>](#) class that contains serialized data.

C#

```
protected SortedSet(System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

The object that contains the information that is required to serialize the `SortedSet<T>` object.

context `StreamingContext`

The structure that contains the source and destination of the serialized stream associated with the `SortedSet<T>` object.

Remarks

This constructor is called during deserialization to reconstitute an object that is transmitted over a stream.

Applies to

- ▼ .NET 10 and other versions

Product	Versions (<i>Obsolete</i>)
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7 (8, 9, 10)
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

SortedSet<T>.Comparer Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the [IComparer<T>](#) object that is used to order the values in the [SortedSet<T>](#).

C#

```
public System.Collections.Generic.IComparer<T> Comparer { get; }
```

Property Value

[IComparer<T>](#)

The comparer that is used to order the values in the [SortedSet<T>](#).

Remarks

The returned comparer can be either the default comparer of the type for a [SortedSet<T>](#), or the comparer used for its constructor.

Retrieving the value of this property is an [\$O\(1\)\$](#) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements in the [SortedSet<T>](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements in the [SortedSet<T>](#).

Implements

[Count](#) , [Count](#) , [Count](#)

Remarks

Retrieving the value of this property is an $O(1)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Max Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the maximum value in the [SortedSet<T>](#), as defined by the comparer.

C#

```
public T? Max { get; }
```

Property Value

T

The maximum value in the set.

Remarks

If the [SortedSet<T>](#) has no elements, then the [Max](#) property returns the default value of `T`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Min Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the minimum value in the [SortedSet<T>](#), as defined by the comparer.

C#

```
public T? Min { get; }
```

Property Value

T

The minimum value in the set.

Remarks

If the [SortedSet<T>](#) has no elements, then the [Min](#) property returns the default value of `T`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an element to the set and returns a value that indicates if it was successfully added.

C#

```
public bool Add(T item);
```

Parameters

item T

The element to add to the set.

Returns

[Boolean](#)

`true` if `item` is added to the set; otherwise, `false`.

Implements

[Add\(T\)](#)

Examples

The following example adds elements to a sorted set. This code example is part of a larger example provided for the [SortedSet<T>](#) class.

C#

```
foreach (string f in files1)
{
    mediaFiles1.Add(f.Substring(f.LastIndexOf(@"\") + 1));
```

Remarks

The `SortedSet<T>` class does not accept duplicate elements. If `item` is already in the set, this method returns `false` and does not throw an exception.

If `Count` already equals the capacity of the `SortedSet<T>` object, the capacity is automatically adjusted to accommodate the new item.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements from the set.

C#

```
public virtual void Clear();
```

Implements

[Clear\(\)](#)

Remarks

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the set contains a specific element.

C#

```
public virtual bool Contains(T item);
```

Parameters

item T

The element to locate in the set.

Returns

[Boolean](#)

`true` if the set contains `item`; otherwise, `false`.

Implements

[Contains\(T\)](#) , [Contains\(T\)](#)

Remarks

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

SortedSet<T>.CopyTo Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies a portion or all of a [SortedSet<T>](#) to a compatible one-dimensional array, starting at the beginning of the destination array or at a specified index.

Overloads

 [Expand table](#)

CopyTo(T[], Int32, Int32)	Copies a specified number of elements from SortedSet<T> to a compatible one-dimensional array, starting at the specified array index.
CopyTo(T[], Int32)	Copies the complete SortedSet<T> to a compatible one-dimensional array, starting at the specified array index.
CopyTo(T[])	Copies the complete SortedSet<T> to a compatible one-dimensional array, starting at the beginning of the target array.

Remarks

This method is an $O(n)$ operation, where n is [Count](#).

CopyTo(T[], Int32, Int32)

Copies a specified number of elements from [SortedSet<T>](#) to a compatible one-dimensional array, starting at the specified array index.

C#

```
public void CopyTo(T[] array, int index, int count);
```

Parameters

array T[]

A one-dimensional array that is the destination of the elements copied from the [SortedSet<T>](#). The array must have zero-based indexing.

index `Int32`

The zero-based index in `array` at which copying begins.

count `Int32`

The number of elements to copy.

Exceptions

[ArgumentException](#)

The number of elements in the source array is greater than the available space from `index` to the end of the destination array.

[ArgumentNullException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`count` is less than zero.

Remarks

This method is an `O(n)` operation, where `n` is `count`.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

CopyTo(T[], Int32)

Copies the complete [SortedSet<T>](#) to a compatible one-dimensional array, starting at the specified array index.

C#

```
public void CopyTo(T[] array, int index);
```

Parameters

array `T[]`

A one-dimensional array that is the destination of the elements copied from the [SortedSet<T>](#). The array must have zero-based indexing.

index `Int32`

The zero-based index in `array` at which copying begins.

Implements

[CopyTo\(T\[\], Int32\)](#)

Exceptions

[ArgumentException](#)

The number of elements in the source array is greater than the available space from `index` to the end of the destination array.

[ArgumentNullException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#)

`index` is less than zero.

Remarks

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

CopyTo(T[])

Copies the complete [SortedSet<T>](#) to a compatible one-dimensional array, starting at the beginning of the target array.

C#

```
public void CopyTo(T[] array);
```

Parameters

array T[]

A one-dimensional array that is the destination of the elements copied from the [SortedSet<T>](#).

Exceptions

[ArgumentException](#)

The number of elements in the source [SortedSet<T>](#) exceeds the number of elements that the destination array can contain.

[ArgumentNullException](#)

array is **null**.

Remarks

The indexing of **array** must be zero-based.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.CreateSetComparer Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an [IEqualityComparer](#) object that can be used to create a collection that contains individual sets.

Overloads

 [Expand table](#)

CreateSetComparer()	Returns an IEqualityComparer object that can be used to create a collection that contains individual sets.
CreateSetComparer(IEqualityComparer<T>)	Returns an IEqualityComparer object, according to a specified comparer, that can be used to create a collection that contains individual sets.

CreateSetComparer()

Returns an [IEqualityComparer](#) object that can be used to create a collection that contains individual sets.

C#

```
public static  
System.Collections.Generic.IEqualityComparer<System.Collections.Generic.SortedS  
et<T>> CreateSetComparer();
```

Returns

[IEqualityComparer<SortedSet<T>>](#)

A comparer for creating a collection of sets.

Remarks

The [IEqualityComparer](#) object checks for equality at only one level; however, you can chain together comparers at additional levels to perform deeper equality testing.

Calling this method is an $O(1)$ operation.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

CreateSetComparer(IEqualityComparer<T>)

Returns an [IEqualityComparer](#) object, according to a specified comparer, that can be used to create a collection that contains individual sets.

C#

```
public static  
System.Collections.Generic.IEqualityComparer<System.Collections.Generic.SortedSet<T>> CreateSetComparer(System.Collections.Generic.IEqualityComparer<T>  
memberEqualityComparer);
```

Parameters

memberEqualityComparer [IEqualityComparer<T>](#)

The comparer to use for creating the returned comparer.

Returns

[IEqualityComparer<SortedSet<T>>](#)

A comparer for creating a collection of sets.

Examples

The following example uses the [CreateSetComparer](#) method to create a set of sets. This code example is part of a larger example provided for the [SortedSet<T>](#) class.

C#

```
// Create a set of the sets.  
IEqualityComparer<SortedSet<string>> comparer =  
    SortedSet<string>.CreateSetComparer();  
  
var allMedia = new HashSet<SortedSet<string>>(comparer);  
allMedia.Add(mediaFiles1);  
allMedia.Add(mediaFiles2);
```

Remarks

The `memberEqualityComparer` and the current [SortedSet<T>](#) must have the same definition of equality.

You can use the comparer returned by this method in the [SortedSet<T>.SortedSet<T>\(IEnumerable<T>, IComparer<T>\)](#) constructor to create a hash table of individual sets.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

SortedSet<T>.ExceptWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements that are in a specified collection from the current [SortedSet<T>](#) object.

C#

```
public void ExceptWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection of items to remove from the [SortedSet<T>](#) object.

Implements

[ExceptWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Examples

The following example removes elements from a sorted set that are duplicated in another sorted set. This code example is part of a larger example provided for the [SortedSet<T>](#) class.

C#

```
// Remove elements in mediaFiles1 that are also in mediaFiles2.
Console.WriteLine("Remove duplicates (of mediaFiles2) from the set...");
Console.WriteLine($"\\tCount before: {mediaFiles1.Count}");
mediaFiles1.ExceptWith(mediaFiles2);
Console.WriteLine($"\\tCount after: {mediaFiles1.Count}");
```

Remarks

This method removes any element in the current `SortedSet<T>` that is also in `other`. Duplicate values in `other` are ignored.

This method is an $O(n \log m)$ operation, where `m` is `Count` and `n` is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the [SortedSet<T>](#).

C#

```
public System.Collections.Generic.SortedSet<T>.Enumerator GetEnumerator();
```

Returns

[SortedSet<T>.Enumerator](#)

An enumerator that iterates through the [SortedSet<T>](#) in sorted order.

Remarks

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.GetObjectData(SerializationInfo, StreamingContext) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface and returns the data that you must have to serialize a [SortedSet<T>](#) object.

C#

```
protected virtual void  
GetObjectData(System.Runtime.Serialization.SerializationInfo info,  
System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information that is required to serialize the [SortedSet<T>](#) object.

context [StreamingContext](#)

A [StreamingContext](#) structure that contains the source and destination of the serialized stream associated with the [SortedSet<T>](#) object.

Exceptions

[ArgumentNullException](#)

`info` is `null`.

Remarks

Calling this method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

SortedSet<T>.GetViewBetween(T, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns a view of a subset in a [SortedSet<T>](#).

C#

```
public virtual System.Collections.Generic.SortedSet<T> GetViewBetween(T?
lowerValue, T? upperValue);
```

Parameters

lowerValue [T](#)

The lowest desired value in the view.

upperValue [T](#)

The highest desired value in the view.

Returns

[SortedSet<T>](#)

A subset view that contains only the values in the specified range.

Exceptions

[ArgumentException](#)

`lowerValue` is more than `upperValue` according to the comparer.

[ArgumentOutOfRangeException](#)

A tried operation on the view was outside the range specified by `lowerValue` and `upperValue`.

Examples

The following example uses the [GetViewBetween](#) method to list only the AVI files from a sorted set of media file names. The comparer evaluates file names according to their extensions. The `lowerValue` is "AVI" and the `upperValue` is only one value higher, "AVJ", to get the view of all AVI files. This code example is part of a larger example provided for the [SortedSet<T>](#) class.

C#

```
// List all the avi files.  
SortedSet<string> aviFiles = mediaFiles1.GetViewBetween("avi", "avj");  
  
Console.WriteLine("AVI files:");  
foreach (string avi in aviFiles)  
{  
    Console.WriteLine($"{avi}");  
}
```

Remarks

This method returns a view of the range of elements that fall between `lowerValue` and `upperValue`, as defined by the comparer. This method does not copy elements from the [SortedSet<T>](#), but provides a window into the underlying [SortedSet<T>](#) itself. You can make changes in both the view and in the underlying [SortedSet<T>](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.IntersectWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Modifies the current [SortedSet<T>](#) object so that it contains only elements that are also in a specified collection.

C#

```
public virtual void IntersectWith(System.Collections.Generic.IEnumerable<T>
other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Implements

[IntersectWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

This method ignores any duplicate elements in `other`.

If the collection represented by the `other` parameter is a [SortedSet<T>](#) collection with the same equality comparer as the current [SortedSet<T>](#) object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is [Count](#) and `m` is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.IsProperSubsetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [SortedSet<T>](#) object is a proper subset of the specified collection.

C#

```
public bool IsProperSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [SortedSet<T>](#) object is a proper subset of `other`; otherwise, `false`.

Implements

[IsProperSubsetOf\(IEnumerable<T>\)](#), [IsProperSubsetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

An empty set is a proper subset of any other collection. Therefore, this method returns `true` if the collection represented by the current [SortedSet<T>](#) object is empty unless the `other` parameter is also an empty set.

This method always returns `false` if `Count` is greater than or equal to the number of elements in `other`.

If the collection represented by `other` is a `SortedSet<T>` collection with the same equality comparer as the current `SortedSet<T>` object, then this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is `Count` and `m` is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.IsProperSupersetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [SortedSet<T>](#) object is a proper superset of the specified collection.

C#

```
public bool IsProperSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [SortedSet<T>](#) object is a proper superset of `other`; otherwise, `false`.

Implements

[IsProperSupersetOf\(IEnumerable<T>\)](#), [IsProperSupersetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

An empty set is a proper superset of any other collection. Therefore, this method returns `true` if the collection represented by the `other` parameter is empty unless the current [SortedSet<T>](#) collection is also empty.

This method always returns `false` if `Count` is less than or equal to the number of elements in `other`.

If the collection represented by `other` is a `SortedSet<T>` collection with the same equality comparer as the current `SortedSet<T>` object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is the number of elements in `other` and `m` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.IsSubsetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [SortedSet<T>](#) object is a subset of the specified collection.

C#

```
public bool IsSubsetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Returns

[Boolean](#)

`true` if the current [SortedSet<T>](#) object is a subset of `other`; otherwise, `false`.

Implements

[IsSubsetOf\(IEnumerable<T>\)](#), [IsSubsetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

An empty set is a subset of any other collection, including an empty set; therefore, this method returns `true` if the collection represented by the current [SortedSet<T>](#) object is empty, even if the `other` parameter is an empty set.

This method always returns `false` if `Count` is greater than the number of elements in `other`.

If the collection represented by `other` is a `SortedSet<T>` collection with the same equality comparer as the current `SortedSet<T>` object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where n is `Count` and m is the number of elements in `other`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.IsSupersetOf(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether a [SortedSet<T>](#) object is a superset of the specified collection.

C#

```
public bool IsSupersetOf(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [SortedSet<T>](#) object is a superset of `other`; otherwise, `false`.

Implements

[IsSupersetOf\(IEnumerable<T>\)](#), [IsSupersetOf\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

All collections, including the empty set, are supersets of the empty set. Therefore, this method returns `true` if the collection represented by the `other` parameter is empty, even if the current [SortedSet<T>](#) object is empty.

This method always returns `false` if `Count` is less than the number of elements in `other`.

If the collection represented by `other` is a `SortedSet<T>` collection with the same equality comparer as the current `SortedSet<T>` object, this method is an $O(n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where n is the number of elements in `other` and m is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.OnDeserialization(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface, and raises the deserialization event when the deserialization is completed.

C#

```
protected virtual void OnDeserialization(object? sender);
```

Parameters

sender [Object](#)

The source of the deserialization event.

Exceptions

[SerializationException](#)

The [SerializationInfo](#) object associated with the current [SortedSet<T>](#) object is invalid.

Remarks

Calling this method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

SortedSet<T>.Overlaps(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the current [SortedSet<T>](#) object and a specified collection share common elements.

C#

```
public bool Overlaps(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Returns

[Boolean](#)

`true` if the [SortedSet<T>](#) object and `other` share at least one common element; otherwise, `false`.

Implements

[Overlaps\(IEnumerable<T>\)](#) , [Overlaps\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

Any duplicate elements in `other` are ignored.

This method is an $O(n \log m)$ operation, where m is [Count](#) and n is the number of elements in [other](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Remove(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes a specified item from the [SortedSet<T>](#).

C#

```
public bool Remove(T item);
```

Parameters

item [T](#)

The element to remove.

Returns

[Boolean](#)

`true` if the element is found and successfully removed; otherwise, `false`.

Implements

[Remove\(T\)](#)

Remarks

If the [SortedSet<T>](#) object does not contain the specified element, the object remains unchanged and no exception is thrown.

`item` can be `null` for reference types.

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.RemoveWhere(Predicate<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all elements that match the conditions defined by the specified predicate from a [SortedSet<T>](#).

C#

```
public int RemoveWhere(Predicate<T> match);
```

Parameters

match [Predicate<T>](#)

The delegate that defines the conditions of the elements to remove.

Returns

[Int32](#)

The number of elements that were removed from the [SortedSet<T>](#) collection.

Exceptions

[ArgumentNullException](#)

match is **null**.

Examples

The following example removes unwanted elements from a sorted set. This code example is part of a larger example provided for the [SortedSet<T>](#) class.

C#

```
// Defines a predicate delegate to use
// for the SortedSet.RemoveWhere method.
private static bool IsDoc(string s)
{
```

```

    s = s.ToLower();
    return (s.EndsWith(".txt") ||
        s.EndsWith(".xls") ||
        s.EndsWith(".xlsx") ||
        s.EndsWith(".pdf") ||
        s.EndsWith(".doc") ||
        s.EndsWith(".docx"));
}

```

C#

```

// Remove elements that have non-media extensions.
// See the 'IsDoc' method.
Console.WriteLine("Remove docs from the set...");
Console.WriteLine($"\\tCount before: {mediaFiles1.Count}");
mediaFiles1.RemoveWhere(IsDoc);
Console.WriteLine($"\\tCount after: {mediaFiles1.Count}");

```

Remarks

`match` must not modify the `SortedSet<T>`. Doing so can cause unexpected results.

Calling this method is an `O(n)` operation, where `n` is `Count`.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.Reverse Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an [IEnumerable<T>](#) that iterates over the [SortedSet<T>](#) in reverse order.

C#

```
public System.Collections.Generic.IEnumerable<T> Reverse();
```

Returns

[IEnumerable<T>](#)

An enumerator that iterates over the [SortedSet<T>](#) in reverse order.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.SetEquals(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether the current [SortedSet<T>](#) object and the specified collection contain the same elements.

C#

```
public bool SetEquals(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Returns

[Boolean](#)

`true` if the current [SortedSet<T>](#) object is equal to `other`; otherwise, `false`.

Implements

[SetEquals\(IEnumerable<T>\)](#) , [SetEquals\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

This method ignores the order of elements and any duplicate elements in `other`.

If the collection represented by `other` is a `SortedSet<T>` collection with the same equality comparer as the current `SortedSet<T>` object, this method is an $O(\log n)$ operation. Otherwise, this method is an $O(n + m)$ operation, where `n` is the number of elements in `other` and `m` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.SymmetricExceptWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Modifies the current [SortedSet<T>](#) object so that it contains only elements that are present either in the current object or in the specified collection, but not both.

C#

```
public void SymmetricExceptWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Implements

[SymmetricExceptWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

`other` is `null`.

Remarks

Any duplicate elements in `other` are ignored.

If the `other` parameter is a [SortedSet<T>](#) collection with the same equality comparer as the current [SortedSet<T>](#) object, this method is an $O(n \log m)$ operation. Otherwise, this method is an $O(n \log m) + O(n \log n)$ operation, where `n` is the number of elements in `other` and `m` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.TryGetValue(T, T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Searches the set for a given value and returns the equal value it finds, if any.

C#

```
public bool TryGetValue(T equalValue, out T actualValue);
```

Parameters

equalValue **T**

The value to search for.

actualValue **T**

The value from the set that the search found, or the default value of T when the search yielded no match.

Returns

[Boolean](#)

A value indicating whether the search was successful.

Remarks

This can be useful when you want to reuse a previously stored reference instead of a newly constructed one (so that more sharing of references can occur) or to look up a value that has more complete data than the value you currently have, although their comparer functions indicate they are equal.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.7.2, 4.8, 4.8.1
.NET Standard	2.1

SortedSet<T>.UnionWith(IEnumerable<T>) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Modifies the current [SortedSet<T>](#) object so that it contains all elements that are present in either the current object or the specified collection.

C#

```
public void UnionWith(System.Collections.Generic.IEnumerable<T> other);
```

Parameters

other [IEnumerable<T>](#)

The collection to compare to the current [SortedSet<T>](#) object.

Implements

[UnionWith\(IEnumerable<T>\)](#)

Exceptions

[ArgumentNullException](#)

other is `null`.

Remarks

Any duplicate elements in **other** are ignored.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.ICollection<T>.Add(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Adds an item to an [ICollection<T>](#) object.

C#

```
void ICollection<T>.Add(T item);
```

Parameters

item T

The object to add to the [ICollection<T>](#) object.

Implements

[Add\(T\)](#)

Exceptions

[NotSupportedException](#)

The [ICollection<T>](#) is read-only.

Remarks

This member is an explicit interface member implementation. It can be used only when the [SortedSet<T>](#) instance is cast to an [ICollection<T>](#) interface.

If [Count](#) is less than [Capacity](#), this method is an $O(1)$ operation. If the capacity must be increased to accommodate the new element, this method becomes an $O(n)$ operation, where [n](#) is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.ICollection<T>.IsReadOnly Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value that indicates whether a [ICollection](#) is read-only.

C#

```
bool System.Collections.Generic.ICollection<T>.IsReadOnly { get; }
```

Property Value

[Boolean](#)

`true` if the collection is read-only; otherwise, `false`.

Implements

[IsReadOnly](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [SortedSet<T>](#) instance is cast to an [ICollection<T>](#) interface.

A collection that is read-only does not allow the addition, removal, or modification of elements after the collection is created.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

Getting the value of this property is an `O(1)` operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.IEnumerable<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.Generic.IEnumerator<T> IEnumerable<T>.GetEnumerator();
```

Returns

[IEnumerator<T>](#)

An enumerator that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [SortedSet<T>](#) instance is cast to an [IEnumerable<T>](#) interface.

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

SortedSet<T>.ICollection.CopyTo(Array, Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the complete [SortedSet<T>](#) to a compatible one-dimensional array, starting at the specified array index.

C#

```
void ICollection.CopyTo(Array array, int index);
```

Parameters

array [Array](#)

A one-dimensional array that is the destination of the elements copied from the [SortedSet<T>](#). The array must have zero-based indexing.

index [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentException](#)

The number of elements in the source array is greater than the available space from [index](#) to the end of the destination array.

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

Remarks

This member is an explicit interface member implementation. It can be used only when the `SortedSet<T>` instance is cast to an [ICollection](#) interface.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value that indicates whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized; otherwise, `false`.

Implements

[IsSynchronized](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [SortedSet<T>](#) instance is cast to an [ICollection](#) interface.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [Dictionary< TKey, TValue >.KeyCollection](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the [SortedSet<T>](#) instance is cast to an [ICollection](#) interface.

This method is an [O\(1\)](#) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1

Product	Versions
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

SortedSet<T>.IEnumarable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumarable.GetEnumerator();
```

Returns

[IEnumerator](#)

An enumerator that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

This member is an explicit interface member implementation. It can be used only when the `SortedSet<T>` instance is cast to an [IEnumerator](#) interface.

This method is an $O(\log n)$ operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

SortedSet<T>.IDeserializationCallback.OnDeserialization(Object) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [IDeserializationCallback](#) interface, and raises the deserialization event when the deserialization is completed.

C#

```
void IDeserializationCallback.OnDeserialization(object sender);
```

Parameters

sender [Object](#)

The source of the deserialization event.

Implements

[OnDeserialization\(Object\)](#)

Exceptions

[SerializationException](#)

The [SerializationInfo](#) object associated with the current [SortedSet<T>](#) instance is invalid.

Remarks

This member is an explicit interface member implementation. It can be used only when the [SortedSet<T>](#) instance is cast to an [IDeserializationCallback](#) interface.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

SortedSet<T>.ISerializable.GetObjectData Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Implements the [ISerializable](#) interface, and returns the data that you need to serialize the [SortedSet<T>](#) instance.

C#

```
void ISerializable.GetObjectData(System.Runtime.Serialization.SerializationInfo  
info, System.Runtime.Serialization.StreamingContext context);
```

Parameters

info [SerializationInfo](#)

A [SerializationInfo](#) object that contains the information that is required to serialize the [SortedSet<T>](#) instance.

context [StreamingContext](#)

A [StreamingContext](#) structure that contains the source and destination of the serialized stream associated with the [SortedSet<T>](#) instance.

Implements

[GetObjectData\(SerializationInfo, StreamingContext\)](#)

Exceptions

[ArgumentNullException](#)

`info` is `null`.

Remarks

This member is an explicit interface member implementation. It can be used only when the [SortedSet<T>](#) instance is cast to an [ISerializable](#) interface.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0, 2.1

Stack<T>.Enumerator Struct

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Enumerates the elements of a [Stack<T>](#).

C#

```
public struct Stack<T>.Enumerator : System.Collections.Generic.IEnumerator<T>
```

Type Parameters

T

Inheritance [Object](#) → [ValueType](#) → [Stack<T>.Enumerator](#)

Implements [IEnumerator<T>](#) , [IEnumerator](#) , [IDisposable](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned

`false`, `Current` is undefined. You cannot set `Current` to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to `MoveNext` or `IEnumerator.Reset` throws an `InvalidOperationException`.

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in `System.Collections.Generic` are not synchronized.

Properties

 Expand table

<code>Current</code>	Gets the element at the current position of the enumerator.
----------------------	---

Methods

 Expand table

<code>Dispose()</code>	Releases all resources used by the <code>Stack<T>.Enumerator</code> .
<code>MoveNext()</code>	Advances the enumerator to the next element of the <code>Stack<T></code> .

Explicit Interface Implementations

 Expand table

<code>IEnumerator.Current</code>	Gets the element at the current position of the enumerator.
<code>IEnumerator.Reset()</code>	Sets the enumerator to its initial position, which is before the first element in the collection. This class cannot be inherited.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerable<T>](#)
- [IEnumerator<T>](#)

Stack<T>.Enumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
public T Current { get; }
```

Property Value

T

The element in the [Stack<T>](#) at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

[Current](#) does not move the position of the enumerator, and consecutive calls to [Current](#) return the same object until either [MoveNext](#) or [IEnumerator.Reset](#) is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator](#)

Stack<T>.Enumerator.Dispose Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Releases all resources used by the [Stack<T>.Enumerator](#).

C#

```
public void Dispose();
```

Implements

[Dispose\(\)](#)

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>.Enumerator.MoveNext Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Advances the enumerator to the next element of the [Stack<T>](#).

C#

```
public bool MoveNext();
```

Returns

[Boolean](#)

`true` if the enumerator was successfully advanced to the next element; `false` if the enumerator has passed the end of the collection.

Implements

[MoveNext\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

After an enumerator is created, the enumerator is positioned before the first element in the collection, and the first call to [MoveNext](#) advances the enumerator to the first element of the collection.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is

irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Current](#)

Stack<T>.Enumerator.IEnumerator.Current Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the element at the current position of the enumerator.

C#

```
object? System.Collections.IEnumerator.Current { get; }
```

Property Value

[Object](#)

The element in the collection at the current position of the enumerator.

Implements

[Current](#)

Exceptions

[InvalidOperationException](#)

The enumerator is positioned before the first element of the collection or after the last element.

Remarks

[IEnumerator.Current](#) is undefined under any of the following conditions:

- The enumerator is positioned before the first element of the collection. That happens after an enumerator is created or after the [IEnumerator.Reset](#) method is called. The [MoveNext](#) method must be called to advance the enumerator to the first element of the collection before reading the value of the [IEnumerator.Current](#) property.
- The last call to [MoveNext](#) returned `false`, which indicates the end of the collection and that the enumerator is positioned after the last element of the collection.

- The enumerator is invalidated due to changes made in the collection, such as adding, modifying, or deleting elements.

`IEnumerator.Current` does not move the position of the enumerator, and consecutive calls to `IEnumerator.Current` return the same object until either `MoveNext` or `IEnumerator.Reset` is called.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator.Reset\(\)](#)

Stack<T>.Enumerator.IEnumerator.Reset Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Sets the enumerator to its initial position, which is before the first element in the collection.
This class cannot be inherited.

C#

```
void IEnumator.Reset();
```

Implements

[Reset\(\)](#)

Exceptions

[InvalidOperationException](#)

The collection was modified after the enumerator was created.

Remarks

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1

Product	Versions
UWP	10.0

See also

- [MoveNext\(\)](#)
- [IEnumerator.Current](#)
- [Current](#)

Stack<T> Class

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Represents a variable size last-in-first-out (LIFO) collection of instances of the same specified type.

C#

```
public class Stack<T> : System.Collections.Generic.IEnumerable<T>,
    System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.ICollection
```

Type Parameters

T

Specifies the type of elements in the stack.

Inheritance [Object](#) → Stack<T>

Implements [IEnumerable<T>](#) , [IReadOnlyCollection<T>](#) , [ICollection](#) , [IEnumerable](#)

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class. The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

[Stack<T>](#) 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<T>](#) if you need to

access the information in the same order that it is stored in the collection. Use [System.Collections.Generic.Stack<T>](#) if you need to access the information in reverse order.

Use the [System.Collections.Concurrent.ConcurrentStack<T>](#) and [System.Collections.Concurrent.ConcurrentQueue<T>](#) types when you need to access the collection from multiple threads concurrently.

A common use for [System.Collections.Generic.Stack<T>](#) is to preserve variable states during calls to other procedures.

Three main operations can be performed on a [System.Collections.Generic.Stack<T>](#) and its elements:

- [Push](#) inserts an element at the top of the [Stack<T>](#).
- [Pop](#) removes an element from the top of the [Stack<T>](#).
- [Peek](#) returns an element that is at the top of the [Stack<T>](#) but does not remove it from the [Stack<T>](#).

The capacity of a [Stack<T>](#) is the number of elements the [Stack<T>](#) can hold. As elements are added to a [Stack<T>](#), the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling [TrimExcess](#).

If [Count](#) is less than the capacity of the stack, [Push](#) is an O(1) operation. If the capacity needs to be increased to accommodate the new element, [Push](#) becomes an O([n](#)) operation, where [n](#) is [Count](#). [Pop](#) is an O(1) operation.

[Stack<T>](#) accepts [null](#) as a valid value for reference types and allows duplicate elements.

Constructors

[] [Expand table](#)

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

Properties

[Expand table](#)

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

Methods

[Expand table](#)

Clear()	Removes all objects from the Stack<T> .
Contains(T)	Determines whether an element is in the Stack<T> .
CopyTo(T[], Int32)	Copies the Stack<T> to an existing one-dimensional Array , starting at the specified array index.
EnsureCapacity(Int32)	Ensures that the capacity of this Stack is at least the specified <code>capacity</code> . If the current capacity is less than <code>capacity</code> , it is increased to at least the specified <code>capacity</code> .
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator for the Stack<T> .
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<T> without removing it.
Pop()	Removes and returns the object at the top of the Stack<T> .
Push(T)	Inserts an object at the top of the Stack<T> .
ToArray()	Copies the Stack<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 Stack<T> , if that number is less than 90 percent of current capacity.

TryPeek(T)	Returns a value that indicates whether there is an object at the top of the Stack<T> , and if one is present, copies it to the <code>result</code> parameter. The object is not removed from the Stack<T> .
TryPop(T)	Returns a value that indicates whether there is an object at the top of the Stack<T> , and if one is present, copies it to the <code>result</code> parameter, and removes it from the Stack<T> .

Explicit Interface Implementations

[] [Expand table](#)

ICollection.CopyTo(Array, Int32)	Copies the elements of the ICollection to an Array , starting at a particular Array index.
ICollection.IsSynchronized	Gets a value indicating whether access to the ICollection is synchronized (thread safe).
ICollection.SyncRoot	Gets an object that can be used to synchronize access to the ICollection .
IEnumerable.GetEnumerator()	Returns an enumerator that iterates through a collection.
IEnumerable<T>.Get Enumerator()	Returns an enumerator that iterates through the collection.

Extension Methods

[] [Expand table](#)

TolImmutableArray<TSource>(IEnumerable<TSource>)	Creates an immutable array from the specified collection.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Constructs an immutable dictionary based on some transformation of a sequence.
TolImmutableDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Constructs an immutable dictionary from an existing collection of elements, applying a transformation function to the source keys.
TolImmutableDictionary<TSource,TKey, TValue>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource, TValue>)	Enumerates and transforms a sequence, and produces an immutable dictionary of its

<code>Func<TSource,TValue>, IEqualityComparer< TKey >, IEqualityComparer< TValue ></code>	contents by using the specified key and value comparers.
<code>ToImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IEqualityComparer< TKey >)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents by using the specified key comparer.
<code>ToImmutableDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable dictionary of its contents.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer< TSource >)</code>	Enumerates a sequence, produces an immutable hash set of its contents, and uses the specified equality comparer for the set type.
<code>ToImmutableHashSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable hash set of its contents.
<code>ToImmutableList<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable list of its contents.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer< TKey >, IEqualityComparer< TValue >)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key and value comparers.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>, IComparer< TKey >)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents by using the specified key comparer.
<code>ToImmutableSortedDictionary<TSource,TKey,TValue> (IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TValue>)</code>	Enumerates and transforms a sequence, and produces an immutable sorted dictionary of its contents.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>, IComparer< TSource >)</code>	Enumerates a sequence, produces an immutable sorted set of its contents, and uses the specified comparer.
<code>ToImmutableSortedSet<TSource>(IEnumerable<TSource>)</code>	Enumerates a sequence and produces an immutable sorted set

	of its contents.
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption, FillEventHandler)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter T is DataRow .
CopyToDataTable<T>(IEnumerable<T>, DataTable, LoadOption)	Copies DataRow objects to the specified DataTable , given an input IEnumerable<T> object where the generic parameter T is DataRow .
CopyToDataTable<T>(IEnumerable<T>)	Returns a DataTable that contains copies of the DataRow objects, given an input IEnumerable<T> object where the generic parameter T is DataRow .
Aggregate<TSource>(IEnumerable<TSource>, Func<TSource,TSource,TSource>)	Applies an accumulator function over a sequence.
Aggregate<TSource,TAccumulate>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value.
Aggregate<TSource,TAccumulate,TResult>(IEnumerable<TSource>, TAccumulate, Func<TAccumulate,TSource,TAccumulate>, Func<TAccumulate,TResult>)	Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value.
All<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether all elements of a sequence satisfy a condition.
Any<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Determines whether any element of a sequence satisfies a condition.
Any<TSource>(IEnumerable<TSource>)	Determines whether a sequence contains any elements.
Append<TSource>(IEnumerable<TSource>, TSource)	Appends a value to the end of the sequence.
AsEnumerable<TSource>(IEnumerable<TSource>)	Returns the input typed as IEnumerable<T> .
Average<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the average of a sequence of Decimal values that are obtained by invoking a

	transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the average of a sequence of Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the average of a sequence of Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the average of a sequence of Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the average of a sequence of nullable Decimal values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)	Computes the average of a sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)	Computes the average of a sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)	Computes the average of a sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)	Computes the average of a sequence of nullable Single values that are obtained by invoking a

	transform function on each element of the input sequence.
Average<TSource>(IEnumerable<TSource>, Func<TSource,Single>)	Computes the average of a sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
Cast<TResult>(IEnumerable)	Casts the elements of an IEnumerable to the specified type.
Chunk<TSource>(IEnumerable<TSource>, Int32)	Splits the elements of a sequence into chunks of size at most size .
Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Concatenates two sequences.
Contains<TSource>(IEnumerable<TSource>, TSource, IEqualityComparer<TSource>)	Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T> .
Contains<TSource>(IEnumerable<TSource>, TSource)	Determines whether a sequence contains a specified element by using the default equality comparer.
Count<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns a number that represents how many elements in the specified sequence satisfy a condition.
Count<TSource>(IEnumerable<TSource>)	Returns the number of elements in a sequence.
DefaultIfEmpty<TSource>(IEnumerable<TSource>, TSource)	Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.
DefaultIfEmpty<TSource>(IEnumerable<TSource>)	Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.

Distinct<TSource>(IEnumerable<TSource>)	Returns distinct elements from a sequence by using the default equality comparer to compare values.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Returns distinct elements from a sequence according to a specified key selector function and using a specified comparer to compare keys.
DistinctBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Returns distinct elements from a sequence according to a specified key selector function.
ElementAt<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence.
ElementAt<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
ElementAtOrDefault<TSource>(IEnumerable<TSource>, Int32)	Returns the element at a specified index in a sequence or a default value if the index is out of range.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set difference of two sequences by using the specified <code>IEqualityComparer<T></code> to compare values.
Except<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set difference of two sequences by using the default equality comparer to compare values.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set difference of two sequences according to a specified key selector function.
ExceptBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)	Produces the set difference of two sequences according to a specified key selector function.
First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the first element in a sequence that satisfies a specified condition.

<code>First<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the first element of a sequence, or a specified default value if the sequence contains no elements.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the first element of the sequence that satisfies a condition or a default value if no such element is found.
<code>FirstOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the first element of a sequence, or a default value if the sequence contains no elements.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.
<code>GroupBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Groups the elements of a sequence according to a specified key selector function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a key selector function. The keys are compared by using a comparer and each group's elements are projected by using a specified function.
<code>GroupBy<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)</code>	Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.
<code>GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>, TResult>, IEqualityComparer<TKey>)</code>	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and

	its key. The keys are compared by using a specified comparer.
GroupBy<TSource,TKey,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TKey,IEnumerable<TSource>,TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource, TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>, TResult>, IEqualityComparer<TKey>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. Key values are compared by using a specified comparer, and the elements of each group are projected by using a specified function.
GroupBy<TSource,TKey,TElement,TResult>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, Func<TKey,IEnumerable<TElement>,TResult>)	Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)	Correlates the elements of two sequences based on key equality and groups the results. A specified IEqualityComparer<T> is used to compare keys.
GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set intersection of two sequences by using the specified IEqualityComparer<T> to compare values.
Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set intersection of two sequences by using the default equality comparer to compare values.
IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set intersection of two sequences according to a

<code>Comparer<TKey></code>	specified key selector function.
<code>IntersectBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TKey>, Func<TSource,TKey>)</code>	Produces the set intersection of two sequences according to a specified key selector function.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)</code>	Correlates the elements of two sequences based on matching keys. A specified <code>IEqualityComparer<T></code> is used to compare keys.
<code>Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)</code>	Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys.
<code>Last<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a specified condition.
<code>Last<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, TSource)</code>	Returns the last element of a sequence, or a specified default value if the sequence contains no elements.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)</code>	Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns the last element of a sequence that satisfies a condition or a default value if no such element is found.
<code>LastOrDefault<TSource>(IEnumerable<TSource>)</code>	Returns the last element of a sequence, or a default value if the sequence contains no elements.
<code>LongCount<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)</code>	Returns an <code>Int64</code> that represents how many elements in a sequence satisfy a condition.
<code>LongCount<TSource>(IEnumerable<TSource>)</code>	Returns an <code>Int64</code> that represents the total number of elements in a sequence.

<code>Max<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Decimal value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Double value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int32 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int64>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Int64 value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Single>>)</code>	Invokes a transform function on each element of a sequence and returns the maximum nullable Single value.
<code>Max<TSource>(IEnumerable<TSource>, Func<TSource,Single>)</code>	Invokes a transform function on each element of a sequence and returns the maximum Single value.

<code>Max<TSource>(IEnumerable<TSource>)</code>	Returns the maximum value in a generic sequence.
<code>Max<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Invokes a transform function on each element of a generic sequence and returns the maximum resulting value.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IComparer<TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function and key comparer.
<code>MaxBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Returns the maximum value in a generic sequence according to a specified key selector function.
<code>Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)</code>	Returns the minimum value in a generic sequence.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Decimal</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Double>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Double</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Int32</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)</code>	Invokes a transform function on each element of a sequence and returns the minimum <code>Int64</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable <code>Decimal</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Double>>)</code>	Invokes a transform function on each element of a sequence and returns the minimum nullable <code>Double</code> value.
<code>Min<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Int32>>)</code>	Invokes a transform function on each element of a sequence and

	returns the minimum nullable Int32 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Int64 value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)	Invokes a transform function on each element of a sequence and returns the minimum nullable Single value.
Min<TSource>(IEnumerable<TSource>, Func<TSource, Single>)	Invokes a transform function on each element of a sequence and returns the minimum Single value.
Min<TSource>(IEnumerable<TSource>)	Returns the minimum value in a generic sequence.
Min<TSource, TResult>(IEnumerable<TSource>, Func<TSource, TResult>)	Invokes a transform function on each element of a generic sequence and returns the minimum resulting value.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function and key comparer.
MinBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Returns the minimum value in a generic sequence according to a specified key selector function.
OfType<TResult>(IEnumerable)	Filters the elements of an IEnumerable based on a specified type.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in ascending order by using a specified comparer.
OrderBy<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>)	Sorts the elements of a sequence in ascending order according to a key.
OrderByDescending<TSource, TKey>(IEnumerable<TSource>, Func<TSource, TKey>, IComparer<TKey>)	Sorts the elements of a sequence in descending order by using a specified comparer.

<code>OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)</code>	Sorts the elements of a sequence in descending order according to a key.
<code>Prepend<TSource>(IEnumerable<TSource>, TSource)</code>	Adds a value to the beginning of the sequence.
<code>Reverse<TSource>(IEnumerable<TSource>)</code>	Inverts the order of the elements in a sequence.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)</code>	Projects each element of a sequence into a new form.
<code>Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,TResult>)</code>	Projects each element of a sequence into a new form by incorporating the element's index.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> and flattens the resulting sequences into one sequence.
<code>SelectMany<TSource,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TResult>>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein.
<code>SelectMany<TSource,TCollection,TResult>(IEnumerable<TSource>, Func<TSource,Int32,IEnumerable<TCollection>>, Func<TSource,TCollection,TResult>)</code>	Projects each element of a sequence to an <code>IEnumerable<T></code> , flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element.
<code>SequenceEqual<TSource>(IEqualityComparer<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Determines whether two sequences are equal by comparing

	their elements by using a specified IEqualityComparer<T> .
SequenceEqual<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.
Single<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.
Single<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, TSource)	Returns the only element of a sequence, or a specified default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition.
SingleOrDefault<TSource>(IEnumerable<TSource>)	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
Skip<TSource>(IEnumerable<TSource>, Int32)	Bypasses a specified number of elements in a sequence and then

	returns the remaining elements.
SkipLast<TSource>(IEnumerable<TSource>, Int32)	Returns a new enumerable collection that contains the elements from <code>source</code> with the last <code>count</code> elements of the source collection omitted.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.
SkipWhile<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Decimal>)	Computes the sum of the sequence of <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Double>)	Computes the sum of the sequence of <code>Double</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int32>)	Computes the sum of the sequence of <code>Int32</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Int64>)	Computes the sum of the sequence of <code>Int64</code> values that are obtained by invoking a transform function on each element of the input sequence.
Sum<TSource>(IEnumerable<TSource>, Func<TSource,Nullable<Decimal>>)	Computes the sum of the sequence of nullable <code>Decimal</code> values that are obtained by invoking a transform function on each element of the input sequence.

<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Double>>)</code>	Computes the sum of the sequence of nullable Double values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int32>>)</code>	Computes the sum of the sequence of nullable Int32 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Int64>>)</code>	Computes the sum of the sequence of nullable Int64 values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Nullable<Single>>)</code>	Computes the sum of the sequence of nullable Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Sum<TSource>(IEnumerable<TSource>, Func<TSource, Single>)</code>	Computes the sum of the sequence of Single values that are obtained by invoking a transform function on each element of the input sequence.
<code>Take<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a specified number of contiguous elements from the start of a sequence.
<code>Take<TSource>(IEnumerable<TSource>, Range)</code>	Returns a specified range of contiguous elements from a sequence.
<code>TakeLast<TSource>(IEnumerable<TSource>, Int32)</code>	Returns a new enumerable collection that contains the last <code>count</code> elements from <code>source</code> .
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is true.
<code>TakeWhile<TSource>(IEnumerable<TSource>, Func<TSource, Int32, Boolean>)</code>	Returns elements from a sequence as long as a specified condition is

	true. The element's index is used in the logic of the predicate function.
ToArray<TSource>(IEnumerable<TSource>)	Creates an array from a IEnumerable<T> .
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToDictionary<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to a specified key selector function, a comparer, and an element selector function.
ToDictionary<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Dictionary<TKey, TValue> from an IEnumerable<T> according to specified key selector and element selector functions.
ToHashSet<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)	Creates a HashSet<T> from an IEnumerable<T> using the comparer to compare keys.
ToHashSet<TSource>(IEnumerable<TSource>)	Creates a HashSet<T> from an IEnumerable<T> .
ToList<TSource>(IEnumerable<TSource>)	Creates a List<T> from an IEnumerable<T> .
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function and key comparer.
ToLookup<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	Creates a Lookup<TKey, TElement> from an IEnumerable<T> according to a specified key selector function.

ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>, IEqualityComparer<TKey>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to a specified key selector function, a comparer and an element selector function.
ToLookup<TSource,TKey,TElement>(IEnumerable<TSource>, Func<TSource,TKey>, Func<TSource,TElement>)	Creates a Lookup<TKey,TElement> from an IEnumerable<T> according to specified key selector and element selector functions.
TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	Attempts to determine the number of elements in a sequence without forcing an enumeration.
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>, IEqualityComparer<TSource>)	Produces the set union of two sequences by using a specified IEqualityComparer<T> .
Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>)	Produces the set union of two sequences by using the default equality comparer.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>, IEqualityComparer<TKey>)	Produces the set union of two sequences according to a specified key selector function.
UnionBy<TSource,TKey>(IEnumerable<TSource>, IEnumerable<TSource>, Func<TSource,TKey>)	Produces the set union of two sequences according to a specified key selector function.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>)	Filters a sequence of values based on a predicate.
Where<TSource>(IEnumerable<TSource>, Func<TSource,Int32,Boolean>)	Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)	Produces a sequence of tuples with elements from the two specified sequences.
Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	Produces a sequence of tuples with elements from the three specified sequences.
Zip<TFirst,TSecond,TResult>(IEnumerable<TFirst>, IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)	Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

AsParallel(IEnumerable)	Enables parallelization of a query.
AsParallel<TSource>(IEnumerable<TSource>)	Enables parallelization of a query.
AsQueryable(IEnumerable)	Converts an IEnumerable to an IQueryable .
AsQueryable<TElement>(IEnumerable<TElement>)	Converts a generic IEnumerable<T> to a generic IQueryable<T> .
Ancestors<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the ancestors of every node in the source collection. Only elements that have a matching XName are included in the collection.
Ancestors<T>(IEnumerable<T>)	Returns a collection of elements that contains the ancestors of every node in the source collection.
DescendantNodes<T>(IEnumerable<T>)	Returns a collection of the descendant nodes of every document and element in the source collection.
Descendants<T>(IEnumerable<T>, XName)	Returns a filtered collection of elements that contains the descendant elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Descendants<T>(IEnumerable<T>)	Returns a collection of elements that contains the descendant elements of every element and document in the source collection.
Elements<T>(IEnumerable<T>, XName)	Returns a filtered collection of the child elements of every element and document in the source collection. Only elements that have a matching XName are included in the collection.
Elements<T>(IEnumerable<T>)	Returns a collection of the child elements of every element and document in the source collection.

InDocumentOrder<T>(IEnumerable<T>)	Returns a collection of nodes that contains all nodes in the source collection, sorted in document order.
Nodes<T>(IEnumerable<T>)	Returns a collection of the child nodes of every document and element in the source collection.
Remove<T>(IEnumerable<T>)	Removes every node in the source collection from its parent node.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Thread Safety

Public static (`Shared` in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.

A [Stack<T>](#) can support multiple readers concurrently, as long as the collection is not modified. Even so, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

See also

- [Iterators \(C#\)](#)
- [Iterators \(Visual Basic\)](#)

Stack<T> Constructors

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Initializes a new instance of the [Stack<T>](#) class.

Overloads

[+] [Expand table](#)

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

Stack<T>()

Initializes a new instance of the [Stack<T>](#) class that is empty and has the default initial capacity.

C#

```
public Stack();
```

Examples

The following code example demonstrates this constructor and several methods of the [Stack<T>](#) generic class.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the

stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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  
*/
```

Remarks

The capacity of a [Stack<T>](#) is the number of elements that the [Stack<T>](#) can hold. As elements are added to a [Stack<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Stack<T>](#).

The capacity can be decreased by calling [TrimExcess](#).

This constructor is an O(1) operation.

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>(IEnumerable<T>)

Initializes a new instance of the [Stack<T>](#) class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.

C#

```
public Stack(System.Collections.Generic.IEnumerable<T> collection);
```

Parameters

collection [IEnumerable<T>](#)

The collection to copy elements from.

Exceptions

[ArgumentNullException](#)

`collection` is `null`.

Examples

The following code example demonstrates this constructor and several methods of the [Stack<T>](#) generic class.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/
```

Remarks

The capacity of a [Stack<T>](#) is the number of elements that the [Stack<T>](#) can hold. As elements are added to a [Stack<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Stack<T>](#).

The capacity can be decreased by calling [TrimExcess](#).

The elements are copied onto the [Stack<T>](#) in the same order they are read by the [IEnumerator<T>](#) of the collection.

This constructor is an O(n) operation, where n is the number of elements in `collection`.

See also

- [ICollection<T>](#)
- [IEnumerator<T>](#)

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>(Int32)

Initializes a new instance of the [Stack<T>](#) class that is empty and has the specified initial capacity or the default initial capacity, whichever is greater.

C#

```
public Stack(int capacity);
```

Parameters

capacity Int32

The initial number of elements that the [Stack<T>](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#)

`capacity` is less than zero.

Remarks

The capacity of a [Stack<T>](#) is the number of elements that the [Stack<T>](#) can hold. As elements are added to a [Stack<T>](#), the capacity is automatically increased as required by reallocating the internal array.

If the size of the collection can be estimated, specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Stack<T>](#).

The capacity can be decreased by calling [TrimExcess](#).

This constructor is an O(n) operation, where n is capacity.

Applies to

- ▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>.Count Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets the number of elements contained in the [Stack<T>](#).

C#

```
public int Count { get; }
```

Property Value

[Int32](#)

The number of elements contained in the [Stack<T>](#).

Implements

[Count](#), [Count](#)

Examples

The following code example demonstrates several properties and methods of the [Stack<T>](#) generic class, including the [Count](#) property.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

The capacity of the [Stack<T>](#) is the number of elements that the [Stack<T>](#) can store. [Count](#) is the number of elements that are actually in the [Stack<T>](#).

The capacity is always greater than or equal to [Count](#). If [Count](#) exceeds the capacity while adding elements, the capacity is increased by automatically reallocating the internal array

before copying the old elements and adding the new elements.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>.Clear Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes all objects from the [Stack<T>](#).

C#

```
public void Clear();
```

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class, including the [Clear](#) method.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/
```

Remarks

[Count](#) is set to zero, and references to other objects from elements of the collection are also released.

The capacity remains unchanged. To reset the capacity of the [Stack<T>](#), call [TrimExcess](#). Trimming an empty [Stack<T>](#) sets the capacity of the [Stack<T>](#) to the default capacity.

This method is an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10

Product	Versions
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>.Contains(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Determines whether an element is in the [Stack<T>](#).

C#

```
public bool Contains(T item);
```

Parameters

item [T](#)

The object to locate in the [Stack<T>](#). The value can be `null` for reference types.

Returns

[Boolean](#)

`true` if `item` is found in the [Stack<T>](#); otherwise, `false`.

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class, including the [Contains](#) method.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to

create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

This method determines equality using the default equality comparer `EqualityComparer<T>.Default` for `T`, the type of values in the list.

This method performs a linear search; therefore, this method is an O(n) operation, where n is Count.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Performing Culture-Insensitive String Operations in Collections](#)

Stack<T>.CopyTo(T[], Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [Stack<T>](#) to an existing one-dimensional [Array](#), starting at the specified array index.

C#

```
public void CopyTo(T[] array, int arrayIndex);
```

Parameters

array [T\[\]](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [Stack<T>](#). The [Array](#) must have zero-based indexing.

arrayIndex [Int32](#)

The zero-based index in [array](#) at which copying begins.

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than zero.

[ArgumentException](#)

The number of elements in the source [Stack<T>](#) is greater than the available space from [arrayIndex](#) to the end of the destination [array](#).

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class, including the [CopyTo](#) method.

The code example creates a stack of strings with default capacity and uses the `Push` method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The `Pop` method is used to pop the first string off the stack. The `Peek` method is used to look at the next item on the stack, and then the `Pop` method is used to pop it off.

The `ToArray` method is used to create an array and copy the stack elements to it, then the array is passed to the `Stack<T>` constructor that takes `IEnumerable<T>`, creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the `CopyTo` method is used to copy the array elements beginning at the middle of the array. The `Stack<T>` constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

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

C#

```
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  
*/
```

Remarks

The elements are copied onto the array in last-in-first-out (LIFO) order, similar to the order of the elements returned by a succession of calls to [Pop](#).

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [ToArray\(\)](#)

Stack<T>.EnsureCapacity(Int32) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Ensures that the capacity of this Stack is at least the specified `capacity`. If the current capacity is less than `capacity`, it is increased to at least the specified `capacity`.

C#

```
public int EnsureCapacity(int capacity);
```

Parameters

capacity [Int32](#)

The minimum capacity to ensure.

Returns

[Int32](#)

The new capacity of this stack.

Applies to

Product	Versions
.NET	6, 7, 8, 9, 10

Stack<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator for the [Stack<T>](#).

C#

```
public System.Collections.Generic.Stack<T>.Enumerator GetEnumerator();
```

Returns

[Stack<T>.Enumerator](#)

An [Stack<T>.Enumerator](#) for the [Stack<T>](#).

Examples

The following code example demonstrates that the [Stack<T>](#) generic class is enumerable. The `foreach` statement (`For Each` in Visual Basic) is used to enumerate the stack.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

The `foreach` statement of the C# language (For Each in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [IEnumerator.Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Stack<T>.Enumerator](#)
- [IEnumerator<T>](#)

Stack<T>.Peek Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns the object at the top of the [Stack<T>](#) without removing it.

C#

```
public T Peek();
```

Returns

T

The object at the top of the [Stack<T>](#).

Exceptions

[InvalidOperationException](#)

The [Stack<T>](#) is empty.

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class, including the [Peek](#) method.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to

create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

This method is similar to the [Pop](#) method, but [Peek](#) does not modify the [Stack<T>](#).

If type `T` is a reference type, `null` can be pushed onto the `Stack<T>` as a placeholder, if needed.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Pop\(\)](#)
- [Push\(T\)](#)

Stack<T>.Pop Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Removes and returns the object at the top of the [Stack<T>](#).

C#

```
public T Pop();
```

Returns

T

The object removed from the top of the [Stack<T>](#).

Exceptions

[InvalidOperationException](#)

The [Stack<T>](#) is empty.

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class, including the [Pop](#) method.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to

create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

This method is similar to the [Peek](#) method, but [Peek](#) does not modify the [Stack<T>](#).

If type `T` is a reference type, `null` can be pushed onto the `Stack<T>` as a placeholder, if needed.

`Stack<T>` is implemented as an array. This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Peek\(\)](#)
- [Push\(T\)](#)

Stack<T>.Push(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Inserts an object at the top of the [Stack<T>](#).

C#

```
public void Push(T item);
```

Parameters

item T

The object to push onto the [Stack<T>](#). The value can be `null` for reference types.

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class, including the [Push](#) method.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

`Stack<T>` is implemented as an array.

If `Count` already equals the capacity, the capacity of the `Stack<T>` is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If type `T` is a reference type, `null` can be pushed onto the `Stack<T>` as a placeholder, if needed. It occupies a slot in the stack and is treated like any object.

If [Count](#) is less than the capacity of the stack, [Push](#) is an O(1) operation. If the capacity needs to be increased to accommodate the new element, [Push](#) becomes an O(n) operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [Peek\(\)](#)
- [Pop\(\)](#)

Stack<T>.ToArray Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the [Stack<T>](#) to a new array.

C#

```
public T[] ToArray();
```

Returns

T[]

A new array containing copies of the elements of the [Stack<T>](#).

Examples

The following code example demonstrates several methods of the [Stack<T>](#) generic class, including the [ToArray](#) method.

The code example creates a stack of strings with default capacity and uses the [Push](#) method to push five strings onto the stack. The elements of the stack are enumerated, which does not change the state of the stack. The [Pop](#) method is used to pop the first string off the stack. The [Peek](#) method is used to look at the next item on the stack, and then the [Pop](#) method is used to pop it off.

The [ToArray](#) method is used to create an array and copy the stack elements to it, then the array is passed to the [Stack<T>](#) constructor that takes [IEnumerable<T>](#), creating a copy of the stack with the order of the elements reversed. The elements of the copy are displayed.

An array twice the size of the stack is created, and the [CopyTo](#) method is used to copy the array elements beginning at the middle of the array. The [Stack<T>](#) constructor is used again to create a copy of the stack with the order of elements reversed; thus, the three null elements are at the end.

The [Contains](#) method is used to show that the string "four" is in the first copy of the stack, after which the [Clear](#) method clears the copy and the [Count](#) property shows that the stack is empty.

C#

```
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
*/

```

Remarks

The elements are copied onto the array in last-in-first-out (LIFO) order, similar to the order of the elements returned by a succession of calls to [Pop](#).

This method is an $O(n)$ operation, where n is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [CopyTo\(T\[\]\), Int32](#)
- [Pop\(\)](#)

Stack<T>.TrimExcess Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Overloads

 [Expand table](#)

[TrimExcess\(\)](#) Sets the capacity to the actual number of elements in the [Stack<T>](#), if that number is less than 90 percent of current capacity.

TrimExcess()

Sets the capacity to the actual number of elements in the [Stack<T>](#), if that number is less than 90 percent of current capacity.

C#

```
public void TrimExcess();
```

Remarks

This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection. The cost of reallocating and copying a large [Stack<T>](#) can be considerable, however, so the [TrimExcess](#) method does nothing if the list is at more than 90 percent of capacity. This avoids incurring a large reallocation cost for a relatively small gain.

This method is an $O(n)$ operation, where n is [Count](#).

To reset a [Stack<T>](#) to its initial state, call the [Clear](#) method before calling [TrimExcess](#) method. Trimming an empty [Stack<T>](#) sets the capacity of the [Stack<T>](#) to the default capacity.

See also

- [Clear\(\)](#)
- [Count](#)

Applies to

▼ .NET 10 and other versions

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>.TryPeek(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns a value that indicates whether there is an object at the top of the [Stack<T>](#), and if one is present, copies it to the `result` parameter. The object is not removed from the [Stack<T>](#).

C#

```
public bool TryPeek(out T result);
```

Parameters

result `T`

If present, the object at the top of the [Stack<T>](#); otherwise, the default value of `T`.

Returns

[Boolean](#)

`true` if there is an object at the top of the [Stack<T>](#); `false` if the [Stack<T>](#) is empty.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Stack<T>.TryPop(T) Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns a value that indicates whether there is an object at the top of the [Stack<T>](#), and if one is present, copies it to the `result` parameter, and removes it from the [Stack<T>](#).

C#

```
public bool TryPop(out T result);
```

Parameters

result `T`

If present, the object at the top of the [Stack<T>](#); otherwise, the default value of `T`.

Returns

[Boolean](#)

`true` if there is an object at the top of the [Stack<T>](#); `false` if the [Stack<T>](#) is empty.

Applies to

Product	Versions
.NET	Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Standard	2.1

Stack<T>.IEnumerable<T>.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through the collection.

C#

```
System.Collections.Generic.IEnumerator<T> IEnumerable<T>.GetEnumerator();
```

Returns

[IEnumerator<T>](#)

An [IEnumerator<T>](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. You cannot set [Current](#) to the first element of the collection again; you must create a new enumerator instance instead.

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IEnumerator<T>](#)

Stack<T>.ICollection.CopyTo(Array, Int32)

Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Copies the elements of the [ICollection](#) to an [Array](#), starting at a particular [Array](#) index.

C#

```
void ICollection.CopyTo(Array array, int arrayIndex);
```

Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from [ICollection](#). The [Array](#) must have zero-based indexing.

arrayIndex [Int32](#)

The zero-based index in [array](#) at which copying begins.

Implements

[CopyTo\(Array, Int32\)](#)

Exceptions

[ArgumentNullException](#)

[array](#) is [null](#).

[ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

`array` does not have zero-based indexing.

-or-

The number of elements in the source [ICollection](#) is greater than the available space from `arrayIndex` to the end of the destination `array`.

-or-

The type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`.

Remarks

ⓘ Note

If the type of the source [ICollection](#) cannot be cast automatically to the type of the destination `array`, the non-generic implementations of [ICollection.CopyTo](#) throw [InvalidOperationException](#), whereas the generic implementations throw [ArgumentException](#).

This method is an $O(n)$ operation, where `n` is [Count](#).

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

Stack<T>.ICollection.IsSynchronized Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets a value indicating whether access to the [ICollection](#) is synchronized (thread safe).

C#

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Property Value

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`. In the default implementation of [Stack<T>](#), this property always returns `false`.

Implements

[IsSynchronized](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. In the rare case where enumerations contend with write accesses, you must lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#).

Synchronization is effective only if all threads lock this object before accessing the collection.

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [SyncRoot](#)

Stack<T>.ICollection.SyncRoot Property

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Gets an object that can be used to synchronize access to the [ICollection](#).

C#

```
object System.Collections.ICollection.SyncRoot { get; }
```

Property Value

[Object](#)

An object that can be used to synchronize access to the [ICollection](#). In the default implementation of [Stack<T>](#), this property always returns the current instance.

Implements

[SyncRoot](#)

Remarks

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

Enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

[SyncRoot](#) returns an object that can be used to synchronize access to the [ICollection](#).

Synchronization is effective only if all threads lock this object before accessing the collection.

The following code shows the use of the [SyncRoot](#) property.

C#

```
ICollection ic = ...;  
lock (ic.SyncRoot) {
```

```
// Access the collection.  
}
```

Retrieving the value of this property is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [IsSynchronized](#)

Stack<T>.IEnumerable.GetEnumerator Method

Definition

Namespace: [System.Collections.Generic](#)

Assemblies: netstandard.dll, System.Collections.dll

Returns an enumerator that iterates through a collection.

C#

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

Returns

[IEnumerator](#)

An [IEnumerator](#) that can be used to iterate through the collection.

Implements

[GetEnumerator\(\)](#)

Remarks

The `foreach` statement of the C# language (`For Each` in Visual Basic) hides the complexity of the enumerators. Therefore, using `foreach` is recommended, instead of directly manipulating the enumerator.

Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.

Initially, the enumerator is positioned before the first element in the collection. [Reset](#) also brings the enumerator back to this position. At this position, [Current](#) is undefined. Therefore, you must call [MoveNext](#) to advance the enumerator to the first element of the collection before reading the value of [Current](#).

[Current](#) returns the same object until either [MoveNext](#) or [Reset](#) is called. [MoveNext](#) sets [Current](#) to the next element.

If [MoveNext](#) passes the end of the collection, the enumerator is positioned after the last element in the collection and [MoveNext](#) returns `false`. When the enumerator is at this position, subsequent calls to [MoveNext](#) also return `false`. If the last call to [MoveNext](#) returned `false`, [Current](#) is undefined. To set [Current](#) to the first element of the collection again, you can call [Reset](#) followed by [MoveNext](#).

An enumerator remains valid as long as the collection remains unchanged. If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is irrecoverably invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#).

The enumerator does not have exclusive access to the collection; therefore, enumerating through a collection is intrinsically not a thread-safe procedure. To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration. To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

Default implementations of collections in [System.Collections.Generic](#) are not synchronized.

This method is an O(1) operation.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0

See also

- [GetEnumerator\(\)](#)
- [GetInternalEnumerator\(\)](#)
- [IEnumerator](#)