

# Contents

## System.Collections

[ArrayList](#)

[Adapter](#)

[Add](#)

[AddRange](#)

[ArrayList](#)

[BinarySearch](#)

[Capacity](#)

[Clear](#)

[Clone](#)

[Contains](#)

[CopyTo](#)

[Count](#)

[FixedSize](#)

[GetEnumerator](#)

[GetRange](#)

[IndexOf](#)

[Insert](#)

[InsertRange](#)

[IsFixedSize](#)

[IsReadOnly](#)

[IsSynchronized](#)

[Item\[Int32\]](#)

[LastIndexOf](#)

[ReadOnly](#)

[Remove](#)

[RemoveAt](#)

[RemoveRange](#)

[Repeat](#)

Reverse  
SetRange  
Sort  
Synchronized  
SyncRoot  
ToArray  
TrimToSize

BitArray  
And  
BitArray  
Clone  
CopyTo  
Count  
Get  
GetEnumerator  
ICollection.CopyTo  
ICollection.Count  
ICollection.IsSynchronized  
ICollection.SyncRoot  
IsReadOnly  
IsSynchronized  
Item[Int32]  
LeftShift  
Length  
Not  
Or  
RightShift  
Set  
SetAll  
SyncRoot  
Xor

CaseInsensitiveComparer  
CaseInsensitiveComparer

Compare  
Default  
DefaultInvariant  
CaseInsensitiveHashCodeProvider  
CaseInsensitiveHashCodeProvider  
Default  
DefaultInvariant  
GetHashCode  
CollectionBase  
Capacity  
Clear  
CollectionBase  
Count  
GetEnumerator  
ICollection.CopyTo  
ICollection.IsSynchronized  
ICollection.SyncRoot  
IList.Add  
IList.Contains  
IList.IndexOf  
IList.Insert  
IList.IsFixedSize  
IList.IsReadOnly  
IList.Item[Int32]  
IList.Remove  
InnerList  
List  
OnClear  
OnClearComplete  
OnInsert  
OnInsertComplete  
OnRemove

[OnRemoveComplete](#)

[OnSet](#)

[OnSetComplete](#)

[OnValidate](#)

[RemoveAt](#)

[Comparer](#)

[Compare](#)

[Comparer](#)

[Default](#)

[DefaultInvariant](#)

[GetObjectData](#)

[DictionaryBase](#)

[Clear](#)

[CopyTo](#)

[Count](#)

[Dictionary](#)

[DictionaryBase](#)

[GetEnumerator](#)

[ICollection.IsSynchronized](#)

[ICollection.SyncRoot](#)

[IDictionary.Add](#)

[IDictionary.Contains](#)

[IDictionaryFixedSize](#)

[IDictionary.ReadOnly](#)

[IDictionary.Item\[Object\]](#)

[IDictionary.Keys](#)

[IDictionary.Remove](#)

[IDictionary.Values](#)

[IEnumerable.GetEnumerator](#)

[InnerHashtable](#)

[OnClear](#)

[OnClearComplete](#)

OnGet  
OnInsert  
OnInsertComplete  
OnRemove  
OnRemoveComplete  
OnSet  
OnSetComplete  
OnValidate  
DictionaryEntry  
Deconstruct  
DictionaryEntry  
Key  
Value  
Hashtable  
Add  
Clear  
Clone  
comparer  
Contains  
ContainsKey  
ContainsValue  
CopyTo  
Count  
EqualityComparer  
GetEnumerator  
GetHash  
GetObjectData  
Hashtable  
hcp  
IEnumerable.GetEnumerator  
IsFixedSize  
IsReadOnly

IsSynchronized

Item[Object]

KeyEquals

Keys

OnDeserialization

Remove

Synchronized

SyncRoot

Values

ICollection

CopyTo

Count

IsSynchronized

SyncRoot

IComparer

Compare

IDictionary

Add

Clear

Contains

GetEnumerator

IsFixedSize

IsReadOnly

Item[Object]

Keys

Remove

Values

IDictionaryEnumerator

Entry

Key

Value

IEnumerable

GetEnumerator

[IEnumerator](#)

[Current](#)

[MoveNext](#)

[Reset](#)

[IEqualityComparer](#)

[Equals](#)

[GetHashCode](#)

[IHashCodeProvider](#)

[GetHashCode](#)

[IList](#)

[Add](#)

[Clear](#)

[Contains](#)

[IndexOf](#)

[Insert](#)

[IsFixedSize](#)

[IsReadOnly](#)

[Item\[Int32\]](#)

[Remove](#)

[RemoveAt](#)

[IStructuralComparable](#)

[CompareTo](#)

[IStructuralEquatable](#)

[Equals](#)

[GetHashCode](#)

[Queue](#)

[Clear](#)

[Clone](#)

[Contains](#)

[CopyTo](#)

[Count](#)

[Dequeue](#)

Enqueue  
GetEnumerator  
IsSynchronized  
Peek  
Queue  
Synchronized  
SyncRoot  
ToArray  
TrimToSize  
ReadOnlyCollectionBase  
Count  
GetEnumerator  
ICollection.CopyTo  
ICollection.IsSynchronized  
ICollection.SyncRoot  
InnerList  
ReadOnlyCollectionBase  
SortedList  
Add  
Capacity  
Clear  
Clone  
Contains  
ContainsKey  
ContainsValue  
CopyTo  
Count  
GetByIndex  
GetEnumerator  
GetKey  
GetKeyList  
GetValueList

[IEnumerable.GetEnumerator](#)

[IndexOfKey](#)

[IndexOfValue](#)

[IsFixedSize](#)

[IsReadOnly](#)

[IsSynchronized](#)

[Item\[Object\]](#)

[Keys](#)

[Remove](#)

[RemoveAt](#)

[SetByIndex](#)

[SortedList](#)

[Synchronized](#)

[SyncRoot](#)

[TrimToSize](#)

[Values](#)

[Stack](#)

[Clear](#)

[Clone](#)

[Contains](#)

[CopyTo](#)

[Count](#)

[GetEnumerator](#)

[IsSynchronized](#)

[Peek](#)

[Pop](#)

[Push](#)

[Stack](#)

[Synchronized](#)

[SyncRoot](#)

[ToArray](#)

[StructuralComparisons](#)

[StructuralComparer](#)

## StructuralEqualityComparer

# System.Collections Namespace

The [System.Collections](#) namespace contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.

## Classes

<a href="#">ArrayList</a>	Implements the <a href="#">IList</a> interface using an array whose size is dynamically increased as required.
<a href="#">BitArray</a>	Manages a compact array of bit values, which are represented as Booleans, where <code>true</code> indicates that the bit is on (1) and <code>false</code> indicates the bit is off (0).
<a href="#">CaseInsensitiveComparer</a>	Compares two objects for equivalence, ignoring the case of strings.
<a href="#">CaseInsensitiveHashCodeProvider</a>	Supplies a hash code for an object, using a hashing algorithm that ignores the case of strings.
<a href="#">CollectionBase</a>	Provides the <code>abstract</code> base class for a strongly typed collection.
<a href="#">Comparer</a>	Compares two objects for equivalence, where string comparisons are case-sensitive.
<a href="#">DictionaryBase</a>	Provides the <code>abstract</code> base class for a strongly typed collection of key/value pairs.
<a href="#">Hashtable</a>	Represents a collection of key/value pairs that are organized based on the hash code of the key.
<a href="#">Queue</a>	Represents a first-in, first-out collection of objects.
<a href="#">ReadOnlyCollectionBase</a>	Provides the <code>abstract</code> base class for a strongly typed non-generic read-only collection.
<a href="#">SortedList</a>	Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.

<a href="#">Stack</a>	Represents a simple last-in-first-out (LIFO) non-generic collection of objects.
<a href="#">StructuralComparisons</a>	Provides objects for performing a structural comparison of two collection objects.

## Structs

<a href="#">DictionaryEntry</a>	Defines a dictionary key/value pair that can be set or retrieved.
---------------------------------	---

## Interfaces

<a href="#">ICollection</a>	Defines size, enumerators, and synchronization methods for all nongeneric collections.
<a href="#">IComparer</a>	Exposes a method that compares two objects.
<a href="#">IDictionary</a>	Represents a nongeneric collection of key/value pairs.
<a href="#">IDictionaryEnumerator</a>	Enumerates the elements of a nongeneric dictionary.
<a href="#">IEnumerable</a>	Exposes an enumerator, which supports a simple iteration over a non-generic collection.
<a href="#">IEnumerator</a>	Supports a simple iteration over a non-generic collection.
<a href="#">IEqualityComparer</a>	Defines methods to support the comparison of objects for equality.
<a href="#">IHashCodeProvider</a>	Supplies a hash code for an object, using a custom hash function.
<a href="#"> IList</a>	Represents a non-generic collection of objects that can be individually accessed by index.
<a href="#">IStructuralComparable</a>	Supports the structural comparison of collection objects.

## IStructuralEquatable

Defines methods to support the comparison of objects for structural equality.

# ArrayList ArrayList Class

Implements the [IList](#) interface using an array whose size is dynamically increased as required.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public class ArrayList : ICloneable, System.Collections.IList

type ArrayList = class
    interface IList
    interface ICloneable
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

### Important

We don't recommend that you use the [ArrayList](#) class for new development. Instead, we recommend that you use the generic [List<T>](#) class. The [ArrayList](#) class is designed to hold heterogeneous collections of objects. However, it does not always offer the best performance. Instead, we recommend the following:

- For a heterogeneous collection of objects, use the [List<Object>](#) (in C#) or [List\(Of Object\)](#) (in Visual Basic) type.
- For a homogeneous collection of objects, use the [List<T>](#) class.

See [Performance Considerations](#) in the [List<T>](#) reference topic for a discussion of the relative performance of these classes. See [Non-generic collections shouldn't be used](#) on GitHub for general information on the use of generic instead of non-generic collection types.

The [ArrayList](#) is not guaranteed to be sorted. You must sort the [ArrayList](#) by calling its [Sort](#) method prior to performing operations (such as [BinarySearch](#)) that require the [ArrayList](#) to be sorted. To maintain a collection that is automatically sorted as new elements are added, you can use the [SortedSet<T>](#) class.

The capacity of an [ArrayList](#) is the number of elements the [ArrayList](#) can hold. As elements are added to an [ArrayList](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#) or by setting the [Capacity](#) property explicitly.

For very large [ArrayList](#) objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the [enabled](#) attribute of the configuration element to [true](#) in the run-time environment.

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

The [ArrayList](#) collection accepts [null](#) as a valid value. It also allows duplicate elements.

Using multidimensional arrays as elements in an [ArrayList](#) collection is not supported.

## Constructors

[ArrayList\(\)](#)  
[ArrayList\(\)](#)

Initializes a new instance of the [ArrayList](#) class that is empty and has the default initial capacity.

`ArrayList(ICollection)`

`ArrayList(ICollection)`

Initializes a new instance of the [ArrayList](#) class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied.

`ArrayList(Int32)`

`ArrayList(Int32)`

Initializes a new instance of the [ArrayList](#) class that is empty and has the specified initial capacity.

## Properties

`Capacity`

`Capacity`

Gets or sets the number of elements that the [ArrayList](#) can contain.

`Count`

`Count`

Gets the number of elements actually contained in the [ArrayList](#).

`IsFixedSize`

`IsFixedSize`

Gets a value indicating whether the [ArrayList](#) has a fixed size.

`IsReadOnly`

`IsReadOnly`

Gets a value indicating whether the [ArrayList](#) is read-only.

`IsSynchronized`

`IsSynchronized`

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

`Item[Int32]`

`Item[Int32]`

Gets or sets the element at the specified index.

`SyncRoot`

`SyncRoot`

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

## Methods

`Adapter(IList)`

`Adapter(IList)`

Creates an [ArrayList](#) wrapper for a specific [IList](#).

`Add(Object)`

`Add(Object)`

Adds an object to the end of the [ArrayList](#).

`AddRange(ICollection)`

`AddRange(ICollection)`

Adds the elements of an [ICollection](#) to the end of the [ArrayList](#).

`BinarySearch(Object)`

`BinarySearch(Object)`

Searches the entire sorted [ArrayList](#) for an element using the default comparer and returns the zero-based index of the element.

`BinarySearch(Object, IComparer)`

`BinarySearch(Object, IComparer)`

Searches the entire sorted [ArrayList](#) for an element using the specified comparer and returns the zero-based index of the element.

`BinarySearch(Int32, Int32, Object, IComparer)`

`BinarySearch(Int32, Int32, Object, IComparer)`

Searches a range of elements in the sorted [ArrayList](#) for an element using the specified comparer and returns the zero-based index of the element.

`Clear()`

`Clear()`

Removes all elements from the [ArrayList](#).

`Clone()`

`Clone()`

Creates a shallow copy of the [ArrayList](#).

`Contains(Object)`

`Contains(Object)`

Determines whether an element is in the [ArrayList](#).

`CopyTo(Array)`

`CopyTo(Array)`

Copies the entire [ArrayList](#) to a compatible one-dimensional [Array](#), starting at the beginning of the target array.

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

Copies the entire [ArrayList](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

`CopyTo(Int32, Array, Int32, Int32)`

`CopyTo(Int32, Array, Int32, Int32)`

Copies a range of elements from the [ArrayList](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

`FixedSize(ArrayList)`

`FixedSize(ArrayList)`

Returns an [ArrayList](#) wrapper with a fixed size.

`FixedSize(IList)`

`FixedSize(IList)`

Returns an [IList](#) wrapper with a fixed size.

`GetEnumerator()`

`GetEnumerator()`

Returns an enumerator for the entire [ArrayList](#).

`GetEnumerator(Int32, Int32)`

`GetEnumerator(Int32, Int32)`

Returns an enumerator for a range of elements in the [ArrayList](#).

```
GetRange(Int32, Int32)
```

```
GetRange(Int32, Int32)
```

Returns an [ArrayList](#) which represents a subset of the elements in the source [ArrayList](#).

```
IndexOf(Object)
```

```
IndexOf(Object)
```

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the entire [ArrayList](#).

```
IndexOf(Object, Int32)
```

```
IndexOf(Object, Int32)
```

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the range of elements in the [ArrayList](#) that extends from the specified index to the last element.

```
IndexOf(Object, Int32, Int32)
```

```
IndexOf(Object, Int32, Int32)
```

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the range of elements in the [ArrayList](#) that starts at the specified index and contains the specified number of elements.

```
Insert(Int32, Object)
```

```
Insert(Int32, Object)
```

Inserts an element into the [ArrayList](#) at the specified index.

```
InsertRange(Int32, ICollection)
```

```
InsertRange(Int32, ICollection)
```

Inserts the elements of a collection into the [ArrayList](#) at the specified index.

```
LastIndexOf(Object)
```

```
LastIndexOf(Object)
```

Searches for the specified [Object](#) and returns the zero-based index of the last occurrence within the entire [ArrayList](#).

```
LastIndexOf(Object, Int32)
```

```
LastIndexOf(Object, Int32)
```

Searches for the specified [Object](#) and returns the zero-based index of the last occurrence within the range of elements in the [ArrayList](#) that extends from the first element to the specified index.

```
LastIndexOf(Object, Int32, Int32)
```

```
LastIndexOf(Object, Int32, Int32)
```

Searches for the specified [Object](#) and returns the zero-based index of the last occurrence within the range of elements in the [ArrayList](#) that contains the specified number of elements and ends at the specified index.

```
ReadOnly(ArrayList)
```

```
ReadOnly(ArrayList)
```

Returns a read-only [ArrayList](#) wrapper.

```
ReadOnly(IList)
```

```
ReadOnly(IList)
```

Returns a read-only [IList](#) wrapper.

```
Remove(Object)
```

```
Remove(Object)
```

Removes the first occurrence of a specific object from the [ArrayList](#).

```
RemoveAt(Int32)
```

```
RemoveAt(Int32)
```

Removes the element at the specified index of the [ArrayList](#).

```
RemoveRange(Int32, Int32)
```

```
RemoveRange(Int32, Int32)
```

Removes a range of elements from the [ArrayList](#).

```
Repeat(Object, Int32)
```

```
Repeat(Object, Int32)
```

Returns an [ArrayList](#) whose elements are copies of the specified value.

```
Reverse()
```

```
Reverse()
```

Reverses the order of the elements in the entire [ArrayList](#).

```
Reverse(Int32, Int32)
```

```
Reverse(Int32, Int32)
```

Reverses the order of the elements in the specified range.

```
SetRange(Int32, ICollection)
```

```
SetRange(Int32, ICollection)
```

Copies the elements of a collection over a range of elements in the [ArrayList](#).

```
Sort()
```

```
Sort()
```

Sorts the elements in the entire [ArrayList](#).

```
Sort(IComparer)
```

```
Sort(IComparer)
```

Sorts the elements in the entire [ArrayList](#) using the specified comparer.

```
Sort(Int32, Int32, IComparer)
```

```
Sort(Int32, Int32, IComparer)
```

Sorts the elements in a range of elements in [ArrayList](#) using the specified comparer.

```
Synchronized(ArrayList)
```

```
Synchronized(ArrayList)
```

Returns an [ArrayList](#) wrapper that is synchronized (thread safe).

```
Synchronized(IList)
```

```
Synchronized(IList)
```

Returns an [IList](#) wrapper that is synchronized (thread safe).

```
ToArray()
```

```
ToArray()
```

Copies the elements of the [ArrayList](#) to a new [Object](#) array.

```
ToArray(Type)
```

```
ToArray(Type)
```

Copies the elements of the [ArrayList](#) to a new array of the specified element type.

```
TrimToSize()
```

```
TrimToSize()
```

Sets the capacity to the actual number of elements in the [ArrayList](#).

## 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.

An [ArrayList](#) can support multiple readers concurrently, as long as the collection is not modified. To guarantee the thread safety of the [ArrayList](#), all operations must be done through the wrapper returned by the [Synchronized\(IList\)](#) method.

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 causes 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.

## See Also

[IList](#) [IList](#)

[IList](#) [IList](#)

# ArrayList.Adapter ArrayList.Adapter

## In this Article

Creates an [ArrayList](#) wrapper for a specific [IList](#).

```
public static System.Collections.ArrayList Adapter (System.Collections.IList list);  
static member Adapter : System.Collections.IList -> System.Collections.ArrayList
```

## Parameters

list [IList](#) [IList](#)

The [IList](#) to wrap.

## Returns

[ArrayList](#) [ArrayList](#)

The [ArrayList](#) wrapper around the [IList](#).

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`list` is `null`.

## Remarks

[Adapter](#) does not copy the contents of [IList](#). Instead, it only creates an [ArrayList](#) wrapper around [IList](#); therefore, changes to the [IList](#) also affect the [ArrayList](#).

The [ArrayList](#) class provides generic [Reverse](#), [BinarySearch](#) and [Sort](#) methods. This wrapper can be a means to use those methods on [IList](#); however, performing these generic operations through the wrapper might be less efficient than operations applied directly on the [IList](#).

This method is an O(1) operation.

## Version Compatibility

In the .NET Framework version 1.0 and 1.1, calling the [GetEnumerator\(Int32, Int32\)](#) method overload on the [ArrayList](#) wrapper returned an enumerator that treated the second argument as an upper bound rather than as a count. In the .NET Framework 2.0 the second argument is correctly treated as a count.

## See

[IList](#)[List](#)

## Also

[BinarySearch\(Int32, Int32, Object, IComparer\)](#)[BinarySearch\(Int32, Int32, Object, IComparer\)](#)  
[Reverse\(\)](#)[Reverse\(\)](#)  
[Sort\(\)](#)[Sort\(\)](#)

# ArrayList.Add ArrayList.Add

## In this Article

Adds an object to the end of the [ArrayList](#).

```
public virtual int Add (object value);  
  
abstract member Add : obj -> int  
override this.Add : obj -> int
```

## Parameters

**value** [Object](#) [Object](#)

The [Object](#) to be added to the end of the [ArrayList](#). The value can be `null`.

## Returns

[Int32](#) [Int32](#)

The [ArrayList](#) index at which the `value` has been added.

## Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to add elements to the [ArrayList](#).

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );

        // Creates and initializes a new Queue.
        Queue myQueue = new Queue();
        myQueue.Enqueue( "jumps" );
        myQueue.Enqueue( "over" );
        myQueue.Enqueue( "the" );
        myQueue.Enqueue( "lazy" );
        myQueue.Enqueue( "dog" );

        // Displays the ArrayList and the Queue.
        Console.WriteLine( "The ArrayList initially contains the following:" );
        PrintValues( myAL, ' ' );
        Console.WriteLine( "The Queue initially contains the following:" );
        PrintValues( myQueue, ' ' );
    }

    // Copies the Queue elements to the end of the ArrayList.
    myAL.AddRange( myQueue );

    // Displays the ArrayList.
    Console.WriteLine( "The ArrayList now contains the following:" );
    PrintValues( myAL, ' ' );
}

public static void PrintValues( IEnumerable myList, char mySeparator ) {
    foreach ( Object obj in myList )
        Console.Write( "{0}{1}", mySeparator, obj );
    Console.WriteLine();
}

/*
This code produces the following output.

The ArrayList initially contains the following:
    The    quick    brown    fox
The Queue initially contains the following:
    jumps    over    the    lazy    dog
The ArrayList now contains the following:
    The    quick    brown    fox    jumps    over    the    lazy    dog
*/

```

## Remarks

[ArrayList](#) accepts `null` as a valid value and allows duplicate elements.

If [Count](#) already equals [Capacity](#), the capacity of the [ArrayList](#) 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](#).

See

[AddRange\(IICollection\)](#)[AddRange\(IICollection\)](#)

Also

[Insert\(Int32, Object\)](#)[Insert\(Int32, Object\)](#)

[Remove\(Object\)](#)[Remove\(Object\)](#)

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

# ArrayList.AddRange ArrayList.AddRange

## In this Article

Adds the elements of an [ICollection](#) to the end of the [ArrayList](#).

```
public virtual void AddRange (System.Collections.ICollection c);  
  
abstract member AddRange : System.Collections.ICollection -> unit  
override this.AddRange : System.Collections.ICollection -> unit
```

## Parameters

c [ICollection](#) [ICollection](#)

The [ICollection](#) whose elements should be added to the end of the [ArrayList](#). The collection itself cannot be `null`, but it can contain elements that are `null`.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`c` is `null`.

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to add elements to the [ArrayList](#).

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );

        // Creates and initializes a new Queue.
        Queue myQueue = new Queue();
        myQueue.Enqueue( "jumps" );
        myQueue.Enqueue( "over" );
        myQueue.Enqueue( "the" );
        myQueue.Enqueue( "lazy" );
        myQueue.Enqueue( "dog" );

        // Displays the ArrayList and the Queue.
        Console.WriteLine( "The ArrayList initially contains the following:" );
        PrintValues( myAL, ' ' );
        Console.WriteLine( "The Queue initially contains the following:" );
        PrintValues( myQueue, ' ' );
    }

    // Copies the Queue elements to the end of the ArrayList.
    myAL.AddRange( myQueue );

    // Displays the ArrayList.
    Console.WriteLine( "The ArrayList now contains the following:" );
    PrintValues( myAL, ' ' );
}

public static void PrintValues( IEnumerable myList, char mySeparator ) {
    foreach ( Object obj in myList )
        Console.Write( "{0}{1}", mySeparator, obj );
    Console.WriteLine();
}

/*
This code produces the following output.

The ArrayList initially contains the following:
    The    quick    brown    fox
The Queue initially contains the following:
    jumps    over    the    lazy    dog
The ArrayList now contains the following:
    The    quick    brown    fox    jumps    over    the    lazy    dog
*/

```

## Remarks

[ArrayList](#) accepts `null` as a valid value and allows duplicate elements.

The order of the elements in the [ICollection](#) is preserved in the [ArrayList](#).

If the new [Count](#) (the current [Count](#) plus the size of the collection) will be greater than [Capacity](#), the capacity of the [ArrayList](#) 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 [ArrayList](#) 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](#).

See

[ICollection](#)[Collection](#)

Also

[Capacity](#)[Capacity](#)

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

[Add\(Object\)](#)[Add\(Object\)](#)

[InsertRange\(Int32, ICollection\)](#)[InsertRange\(Int32, ICollection\)](#)

[SetRange\(Int32, ICollection\)](#)[SetRange\(Int32, ICollection\)](#)

[GetRange\(Int32, Int32\)](#)[GetRange\(Int32, Int32\)](#)

[RemoveRange\(Int32, Int32\)](#)[RemoveRange\(Int32, Int32\)](#)

# ArrayList ArrayList

In this Article

## Overloads

<a href="#">ArrayList()</a>	Initializes a new instance of the <a href="#">ArrayList</a> class that is empty and has the default initial capacity.
<a href="#">ArrayList(ICollection)</a> <a href="#">ArrayList(ICollection)</a>	Initializes a new instance of the <a href="#">ArrayList</a> class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied.
<a href="#">ArrayList(Int32)</a> <a href="#">ArrayList(Int32)</a>	Initializes a new instance of the <a href="#">ArrayList</a> class that is empty and has the specified initial capacity.

## ArrayList()

Initializes a new instance of the [ArrayList](#) class that is empty and has the default initial capacity.

```
public ArrayList ();
```

### Remarks

The capacity of an [ArrayList](#) is the number of elements that the [ArrayList](#) can hold. As elements are added to an [ArrayList](#), 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 [ArrayList](#).

This constructor is an O(1) operation.

See

[Capacity](#)

Also

## ArrayList(ICollection) ArrayList(ICollection)

Initializes a new instance of the [ArrayList](#) class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied.

```
public ArrayList (System.Collections.ICollection c);  
new System.Collections.ArrayList : System.Collections.ICollection -> System.Collections.ArrayList
```

### Parameters

c [ICollection](#)

The [ICollection](#) whose elements are copied to the new list.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`c` is `null`.

## Remarks

The capacity of an [ArrayList](#) is the number of elements that the [ArrayList](#) can hold. As elements are added to an [ArrayList](#), 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 [ArrayList](#).

The elements are copied onto the [ArrayList](#) in the same order they are read by the [IEnumerator](#) of the [ICollection](#).

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

See

[ICollection](#)

Also

[Capacity](#)

## ArrayList(Int32) ArrayList(Int32)

Initializes a new instance of the [ArrayList](#) class that is empty and has the specified initial capacity.

```
public ArrayList (int capacity);  
new System.Collections.ArrayList : int -> System.Collections.ArrayList
```

## Parameters

capacity [Int32](#) [Int32](#)

The number of elements that the new list can initially store.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

## Remarks

The capacity of an [ArrayList](#) is the number of elements that the [ArrayList](#) can hold. As elements are added to an [ArrayList](#), 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 [ArrayList](#).

This constructor is an  $O(n)$  operation, where  $n$  is `capacity`.

See

[Capacity](#)

Also

# ArrayList.BinarySearch ArrayList.BinarySearch

In this Article

## Overloads

<code>BinarySearch(Object) BinarySearch(Object)</code>	Searches the entire sorted <a href="#">ArrayList</a> for an element using the default comparer and returns the zero-based index of the element.
<code>BinarySearch(Object, IComparer) BinarySearch(Object, IComparer)</code>	Searches the entire sorted <a href="#">ArrayList</a> for an element using the specified comparer and returns the zero-based index of the element.
<code>BinarySearch(Int32, Int32, Object, IComparer) BinarySearch(Int32, Int32, Object, IComparer)</code>	Searches a range of elements in the sorted <a href="#">ArrayList</a> for an element using the specified comparer and returns the zero-based index of the element.

## BinarySearch(Object) BinarySearch(Object)

Searches the entire sorted [ArrayList](#) for an element using the default comparer and returns the zero-based index of the element.

```
public virtual int BinarySearch (object value);  
abstract member BinarySearch : obj -> int  
override this.BinarySearch : obj -> int
```

Parameters

`value` [Object](#) [Object](#)

The [Object](#) to locate. The value can be `null`.

Returns

[Int32](#) [Int32](#)

The zero-based index of `value` in the sorted [ArrayList](#), if `value` is found; otherwise, a negative number, which is the bitwise complement of the index of the next element that is larger than `value` or, if there is no larger element, the bitwise complement of [Count](#).

Exceptions

[ArgumentException](#) [ArgumentNullException](#)

Neither `value` nor the elements of [ArrayList](#) implement the [IComparable](#) interface.

[InvalidOperationException](#) [InvalidOperationException](#)

`value` is not of the same type as the elements of the [ArrayList](#).

Examples

The following code example shows how to use [BinarySearch](#) to locate a specific object in the [ArrayList](#).

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList. BinarySearch requires
        // a sorted ArrayList.
        ArrayList myAL = new ArrayList();
        for ( int i = 0; i <= 4; i++ )
            myAL.Add( i*2 );

        // Displays the ArrayList.
        Console.WriteLine( "The Int32 ArrayList contains the following:" );
        PrintValues( myAL );

        // Locates a specific object that does not exist in the ArrayList.
        Object myObjectOdd = 3;
        FindMyObject( myAL, myObjectOdd );

        // Locates an object that exists in the ArrayList.
        Object myObjectEven = 6;
        FindMyObject( myAL, myObjectEven );
    }

    public static void FindMyObject( ArrayList myList, Object myObject ) {
        int myIndex=myList.BinarySearch( myObject );
        if ( myIndex < 0 )
            Console.WriteLine( "The object to search for ({0}) is not found. The next larger object is
at index {1}.", myObject, ~myIndex );
        else
            Console.WriteLine( "The object to search for ({0}) is at index {1}.", myObject, myIndex );
    }

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

/*
This code produces the following output.

The Int32 ArrayList contains the following:
  0  2  4  6  8
The object to search for (3) is not found. The next larger object is at index 2.
The object to search for (6) is at index 3.
*/

```

## Remarks

The `value` parameter and each element of the `ArrayList` must implement the `IComparable` interface, which is used for comparisons. The elements of the `ArrayList` must already be sorted in increasing value according to the sort order defined by the `IComparable` implementation; otherwise, the result might be incorrect.

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

If the `ArrayList` 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 `ArrayList` 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 [ArrayList](#), 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 [Count](#).

See

[Performing Culture-Insensitive String Operations in Collections](#)

Also

## BinarySearch([Object](#), [IComparer](#)) BinarySearch([Object](#), [IComparer](#))

Searches the entire sorted [ArrayList](#) for an element using the specified comparer and returns the zero-based index of the element.

```
public virtual int BinarySearch (object value, System.Collections.IComparer comparer);  
abstract member BinarySearch : obj * System.Collections.IComparer -> int  
override this.BinarySearch : obj * System.Collections.IComparer -> int
```

Parameters

value [Object](#) [Object](#)

The [Object](#) to locate. The value can be [null](#).

comparer [IComparer](#) [IComparer](#)

The [IComparer](#) implementation to use when comparing elements.

-or-

[null](#) to use the default comparer that is the [IComparable](#) implementation of each element.

Returns

[Int32](#) [Int32](#)

The zero-based index of [value](#) in the sorted [ArrayList](#), if [value](#) is found; otherwise, a negative number, which is the bitwise complement of the index of the next element that is larger than [value](#) or, if there is no larger element, the bitwise complement of [Count](#).

Exceptions

[ArgumentException](#) [ArgumentException](#)

[comparer](#) is [null](#) and neither [value](#) nor the elements of [ArrayList](#) implement the [IComparable](#) interface.

[InvalidOperationException](#) [InvalidOperationException](#)

[comparer](#) is [null](#) and [value](#) is not of the same type as the elements of the [ArrayList](#).

Examples

The following example creates an [ArrayList](#) of colored animals. The provided [IComparer](#) performs the string comparison for the binary search. The results of both an iterative search and a binary search are displayed.

```

using System;
using System.Collections;

public class SimpleStringComparer : IComparer
{
    int IComparer.Compare(object x, object y)
    {
        string cmpstr = (string)x;
        return cmpstr.CompareTo((string)y);
    }
}

public class MyArrayList : ArrayList
{
    public static void Main()
    {
        // Creates and initializes a new ArrayList.
        MyArrayList coloredAnimals = new MyArrayList();

        coloredAnimals.Add("White Tiger");
        coloredAnimals.Add("Pink Bunny");
        coloredAnimals.Add("Red Dragon");
        coloredAnimals.Add("Green Frog");
        coloredAnimals.Add("Blue Whale");
        coloredAnimals.Add("Black Cat");
        coloredAnimals.Add("Yellow Lion");

        // BinarySearch requires a sorted ArrayList.
        coloredAnimals.Sort();

        // Compare results of an iterative search with a binary search
        int index = coloredAnimals.IterativeSearch("White Tiger");
        Console.WriteLine("Iterative search, item found at index: {0}", index);

        index = coloredAnimals.BinarySearch("White Tiger", new SimpleStringComparer());
        Console.WriteLine("Binary search, item found at index: {0}", index);
    }

    public int IterativeSearch(object finditem)
    {
        int index = -1;

        for (int i = 0; i < this.Count; i++)
        {
            if (finditem.Equals(this[i]))
            {
                index = i;
                break;
            }
        }
        return index;
    }
}
// This code produces the following output.
//
// Iterative search, item found at index: 5
// Binary search, item found at index: 5
//

```

## 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 `ArrayList` are compared to the specified value using the specified `IComparer` implementation. The elements of the `ArrayList` must already be sorted in increasing value according to the sort order defined by `comparer`; otherwise, the result might be incorrect.

If `comparer` is `null`, the comparison is done using the `IComparable` implementation provided by the element itself or by the specified value. The elements of the `ArrayList` must already be sorted in increasing value according to the sort order defined by the `IComparable` implementation; otherwise, the result might be incorrect.

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

If the `ArrayList` 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 `ArrayList` 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 `ArrayList`, 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 `Count`.

See

[Performing Culture-Insensitive String Operations in Collections](#)

Also

## BinarySearch(Int32, Int32, Object, IComparer) BinarySearch(Int32, Int32, Object, IComparer)

Searches a range of elements in the sorted `ArrayList` for an element using the specified comparer and returns the zero-based index of the element.

```
public virtual int BinarySearch (int index, int count, object value, System.Collections.IComparer comparer);  
  
abstract member BinarySearch : int * int * obj * System.Collections.IComparer -> int  
override this.BinarySearch : int * int * obj * System.Collections.IComparer -> int
```

### Parameters

index `Int32` `Int32`

The zero-based starting index of the range to search.

count `Int32` `Int32`

The length of the range to search.

value `Object` `Object`

The `Object` to locate. The value can be `null`.

comparer `IComparer` `IComparer`

The `IComparer` implementation to use when comparing elements.

-or-

`null` to use the default comparer that is the `IComparable` implementation of each element.

### Returns

## Int32 Int32

The zero-based index of `value` in the sorted `ArrayList`, if `value` is found; otherwise, a negative number, which is the bitwise complement of the index of the next element that is larger than `value` or, if there is no larger element, the bitwise complement of `Count`.

Exceptions

### ArgumentException ArgumentException

`index` and `count` do not denote a valid range in the `ArrayList`.

-or-

`comparer` is `null` and neither `value` nor the elements of `ArrayList` implement the `IComparable` interface.

### InvalidOperationException InvalidOperationException

`comparer` is `null` and `value` is not of the same type as the elements of the `ArrayList`.

### ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than zero.

-or-

`count` is less than zero.

## 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 `ArrayList` are compared to the specified value using the specified `IComparer` implementation. The elements of the `ArrayList` must already be sorted in increasing value according to the sort order defined by `comparer`; otherwise, the result might be incorrect.

If `comparer` is `null`, the comparison is done using the `IComparable` implementation provided by the element itself or by the specified value. The elements of the `ArrayList` must already be sorted in increasing value according to the sort order defined by the `IComparable` implementation; otherwise, the result might be incorrect.

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

If the `ArrayList` 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 `ArrayList` 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 `ArrayList`, 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 `count`.

See

`IComparer`

Also

`Performing Culture-Insensitive String Operations in Collections`

# ArrayList.Capacity

## In this Article

Gets or sets the number of elements that the [ArrayList](#) can contain.

```
public virtual int Capacity { get; set; }  
member this.Capacity : int with get, set
```

Returns

[Int32](#)

The number of elements that the [ArrayList](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[Capacity](#) is set to a value that is less than [Count](#).

[OutOfMemoryException](#) [OutOfMemoryException](#)

There is not enough memory available on the system.

## Remarks

[Capacity](#) is the number of elements that the [ArrayList](#) can store. [Count](#) is the number of elements that are actually in the [ArrayList](#).

[Capacity](#) is always greater than or equal to [Count](#). If [Count](#) exceeds [Capacity](#) while adding elements, the capacity is automatically increased by reallocating the internal array before copying the old elements and adding the new elements.

The capacity can be decreased by calling [TrimToSize](#) 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.

See

[Count](#)

Also

# ArrayList.Clear ArrayList.Clear

## In this Article

Removes all elements from the [ArrayList](#).

```
public virtual void Clear ();  
  
abstract member Clear : unit -> unit  
override this.Clear : unit -> unit
```

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to trim the unused portions of the [ArrayList](#) and how to clear the values of the [ArrayList](#).

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "The" );  
        myAL.Add( "quick" );  
        myAL.Add( "brown" );  
        myAL.Add( "fox" );  
        myAL.Add( "jumps" );  
  
        // Displays the count, capacity and values of the ArrayList.  
        Console.WriteLine( "Initially," );  
        Console.WriteLine( "    Count    : {0}", myAL.Count );  
        Console.WriteLine( "    Capacity : {0}", myAL.Capacity );  
        Console.Write( "    Values:" );  
        PrintValues( myAL );  
  
        // Trim the ArrayList.  
        myAL.TrimToSize();  
  
        // Displays the count, capacity and values of the ArrayList.  
        Console.WriteLine( "After TrimToSize," );  
        Console.WriteLine( "    Count    : {0}", myAL.Count );  
        Console.WriteLine( "    Capacity : {0}", myAL.Capacity );  
        Console.Write( "    Values:" );  
        PrintValues( myAL );  
  
        // Clear the ArrayList.  
        myAL.Clear();  
  
        // Displays the count, capacity and values of the ArrayList.  
        Console.WriteLine( "After Clear," );  
        Console.WriteLine( "    Count    : {0}", myAL.Count );  
        Console.WriteLine( "    Capacity : {0}", myAL.Capacity );
```

```

        Console.Write( "    Values:" );
        PrintValues( myAL );

        // Trim the ArrayList again.
        myAL.TrimToSize();

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

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

/*
This code produces the following output.

Initially,
Count      : 5
Capacity   : 16
Values:    The    quick    brown    fox    jumps
After TrimToSize,
Count      : 5
Capacity   : 5
Values:    The    quick    brown    fox    jumps
After Clear,
Count      : 0
Capacity   : 5
Values:
After the second TrimToSize,
Count      : 0
Capacity   : 16
Values:
*/

```

## 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 [ArrayList](#), call [TrimToSize](#) or set the [Capacity](#) property directly. Trimming an empty [ArrayList](#) sets the capacity of the [ArrayList](#) to the default capacity.

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

See

[TrimToSize\(\)](#)  
[TrimToSize\(\)](#)

Also

[Capacity](#)  
[Capacity](#)  
[Count](#)  
[Count](#)

# ArrayList.Clone ArrayList.Clone

## In this Article

Creates a shallow copy of the [ArrayList](#).

```
public virtual object Clone ();  
  
abstract member Clone : unit -> obj  
override this.Clone : unit -> obj
```

Returns

[Object Object](#)

A shallow copy of the [ArrayList](#).

## Remarks

A shallow copy of a collection copies only the elements of the collection, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new collection point to the same objects that the references in the original collection point to.

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](#).

# ArrayList.Contains ArrayList.Contains

## In this Article

Determines whether an element is in the [ArrayList](#).

```
public virtual bool Contains (object item);  
  
abstract member Contains : obj -> bool  
override this.Contains : obj -> bool
```

## Parameters

item Object Object

The [Object](#) to locate in the [ArrayList](#). The value can be `null`.

## Returns

[Boolean Boolean](#)

`true` if `item` is found in the [ArrayList](#); otherwise, `false`.

## Remarks

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where `n` is [Count](#).

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether `item` exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

## See

[IndexOf\(Object\)IndexOf\(Object\)](#)

## Also

[LastIndexOf\(Object\)LastIndexOf\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# ArrayList.CopyTo ArrayList.CopyTo

In this Article

## Overloads

<a href="#">CopyTo(Array)</a> <a href="#">CopyTo(Array)</a>	Copies the entire <a href="#">ArrayList</a> to a compatible one-dimensional <a href="#">Array</a> , starting at the beginning of the target array.
<a href="#">CopyTo(Array, Int32)</a> <a href="#">CopyTo(Array, Int32)</a>	Copies the entire <a href="#">ArrayList</a> to a compatible one-dimensional <a href="#">Array</a> , starting at the specified index of the target array.
<a href="#">CopyTo(Int32, Array, Int32, Int32)</a> <a href="#">CopyTo(Int32, Array, Int32, Int32)</a>	Copies a range of elements from the <a href="#">ArrayList</a> to a compatible one-dimensional <a href="#">Array</a> , starting at the specified index of the target array.

## CopyTo(Array) CopyTo(Array)

Copies the entire [ArrayList](#) to a compatible one-dimensional [Array](#), starting at the beginning of the target array.

```
public virtual void CopyTo (Array array);  
abstract member CopyTo : Array -> unit  
override this.CopyTo : Array -> unit
```

### Parameters

array [Array](#) [Array](#)

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

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`array` is `null`.

[ArgumentException](#) [ArgumentException](#)

`array` is multidimensional.

-or-

The number of elements in the source [ArrayList](#) is greater than the number of elements that the destination `array` can contain.

[InvalidCastException](#) [InvalidCastException](#)

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

### Examples

The following code example shows how to copy an [ArrayList](#) into a one-dimensional [System.Array](#).

```
using System;
```

```

using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes the source ArrayList.
        ArrayList mySourceList = new ArrayList();
        mySourceList.Add( "three" );
        mySourceList.Add( "napping" );
        mySourceList.Add( "cats" );
        mySourceList.Add( "in" );
        mySourceList.Add( "the" );
        mySourceList.Add( "barn" );

        // Creates and initializes the one-dimensional target Array.
        String[] myTargetArray = new String[15];
        myTargetArray[0] = "The";
        myTargetArray[1] = "quick";
        myTargetArray[2] = "brown";
        myTargetArray[3] = "fox";
        myTargetArray[4] = "jumps";
        myTargetArray[5] = "over";
        myTargetArray[6] = "the";
        myTargetArray[7] = "lazy";
        myTargetArray[8] = "dog";

        // Displays the values of the target Array.
        Console.WriteLine( "The target Array contains the following (before and after copying):" );
        PrintValues( myTargetArray, ' ' );

        // Copies the second element from the source ArrayList to the target Array starting at index
7.
        mySourceList.CopyTo( 1, myTargetArray, 7, 1 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source ArrayList to the target Array starting at index 6.
        mySourceList.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source ArrayList to the target Array starting at index 0.
        mySourceList.CopyTo( myTargetArray );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );
    }

    public static void PrintValues( String[] myArr, char mySeparator ) {
        for ( int i = 0; i < myArr.Length; i++ )
            Console.Write( "{0}{1}", mySeparator, myArr[i] );
        Console.WriteLine();
    }
}

```

/\*
This code produces the following output.

The target Array contains the following (before and after copying):  
The quick brown fox jumps over the lazy dog  
The quick brown fox jumps over the napping dog  
The quick brown fox jumps over three napping cats in the barn

```
the quick brown fox jumps over three napping cats in the barn  
three napping cats in the barn three napping cats in the barn  
*/
```

## Remarks

The specified array must be of a compatible type.

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 [ArrayList](#).

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

## **CopyTo(Array, Int32) CopyTo(Array, Int32)**

Copies the entire [ArrayList](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

```
public virtual void CopyTo (Array array, int arrayIndex);  
  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

## Parameters

array	<a href="#">Array</a>	<a href="#">Array</a>
-------	-----------------------	-----------------------

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

arrayIndex	<a href="#">Int32</a>	<a href="#">Int32</a>
------------	-----------------------	-----------------------

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

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than zero.

[ArgumentException](#) [ArgumentException](#)

[array](#) is multidimensional.

-or-

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

[InvalidCastException](#) [InvalidCastException](#)

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

## Examples

The following code example shows how to copy an [ArrayList](#) into a one-dimensional [System.Array](#).

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

```

public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes the source ArrayList.
        ArrayList mySourceList = new ArrayList();
        mySourceList.Add( "three" );
        mySourceList.Add( "napping" );
        mySourceList.Add( "cats" );
        mySourceList.Add( "in" );
        mySourceList.Add( "the" );
        mySourceList.Add( "barn" );

        // Creates and initializes the one-dimensional target Array.
        String[] myTargetArray = new String[15];
        myTargetArray[0] = "The";
        myTargetArray[1] = "quick";
        myTargetArray[2] = "brown";
        myTargetArray[3] = "fox";
        myTargetArray[4] = "jumps";
        myTargetArray[5] = "over";
        myTargetArray[6] = "the";
        myTargetArray[7] = "lazy";
        myTargetArray[8] = "dog";

        // Displays the values of the target Array.
        Console.WriteLine( "The target Array contains the following (before and after copying):" );
        PrintValues( myTargetArray, ' ' );

        // Copies the second element from the source ArrayList to the target Array, starting at index
7.
        mySourceList.CopyTo( 1, myTargetArray, 7, 1 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source ArrayList to the target Array, starting at index 6.
        mySourceList.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source ArrayList to the target Array, starting at index 0.
        mySourceList.CopyTo( myTargetArray );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );
    }

    public static void PrintValues( String[] myArr, char mySeparator ) {
        for ( int i = 0; i < myArr.Length; i++ )
            Console.Write( "{0}{1}", mySeparator, myArr[i] );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The target Array contains the following (before and after copying):
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the napping dog

```

```
The quick brown fox jumps over three napping cats in the barn  
three napping cats in the barn three napping cats in the barn
```

```
*/
```

## Remarks

The specified array must be of a compatible type.

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 [ArrayList](#).

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

## **CopyTo(Int32, Array, Int32, Int32) CopyTo(Int32, Array, Int32, Int32)**

Copies a range of elements from the [ArrayList](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

```
public virtual void CopyTo (int index, Array array, int arrayIndex, int count);  
  
abstract member CopyTo : int * Array * int * int -> unit  
override this.CopyTo : int * Array * int * int -> unit
```

## Parameters

index [Int32](#) [Int32](#)

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

array [Array](#) [Array](#)

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

arrayIndex [Int32](#) [Int32](#)

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

count [Int32](#) [Int32](#)

The number of elements to copy.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[index](#) is less than zero.

-or-

[arrayIndex](#) is less than zero.

-or-

[count](#) is less than zero.

## ArgumentException ArgumentException

`array` is multidimensional.

-or-

`index` is equal to or greater than the [Count](#) of the source [ArrayList](#).

-or-

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

## InvalidCastException InvalidCastException

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

### Examples

The following code example shows how to copy an [ArrayList](#) into a one-dimensional [System.Array](#).

```
using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes the source ArrayList.
        ArrayList mySourceList = new ArrayList();
        mySourceList.Add( "three" );
        mySourceList.Add( "napping" );
        mySourceList.Add( "cats" );
        mySourceList.Add( "in" );
        mySourceList.Add( "the" );
        mySourceList.Add( "barn" );

        // Creates and initializes the one-dimensional target Array.
        String[] myTargetArray = new String[15];
        myTargetArray[0] = "The";
        myTargetArray[1] = "quick";
        myTargetArray[2] = "brown";
        myTargetArray[3] = "fox";
        myTargetArray[4] = "jumps";
        myTargetArray[5] = "over";
        myTargetArray[6] = "the";
        myTargetArray[7] = "lazy";
        myTargetArray[8] = "dog";

        // Displays the values of the target Array.
        Console.WriteLine( "The target Array contains the following (before and after copying):" );
        PrintValues( myTargetArray, ' ' );

        // Copies the second element from the source ArrayList to the target Array, starting at index
        // 7.
        mySourceList.CopyTo( 1, myTargetArray, 7, 1 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source ArrayList to the target Array, starting at index 6.
        mySourceList.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );
    }
}
```

```
// Copies the entire source ArrayList to the target Array, starting at index 0.  
mySourceList.CopyTo( myTargetArray );  
  
// Displays the values of the target Array.  
PrintValues( myTargetArray, ' ' );  
  
}  
  
public static void PrintValues( String[] myArr, char mySeparator ) {  
    for ( int i = 0; i < myArr.Length; i++ )  
        Console.Write( "{0}{1}", mySeparator, myArr[i] );  
    Console.WriteLine();  
}  
  
}  
  
/*  
This code produces the following output.  
  
The target Array contains the following (before and after copying):  
The quick brown fox jumps over the lazy dog  
The quick brown fox jumps over the napping dog  
The quick brown fox jumps over three napping cats in the barn  
three napping cats in the barn three napping cats in the barn  
  
*/
```

## Remarks

The specified array must be of a compatible type.

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 [ArrayList](#).

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

# ArrayList.Count ArrayList.Count

## In this Article

Gets the number of elements actually contained in the [ArrayList](#).

```
public virtual int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements actually contained in the [ArrayList](#).

## Remarks

[Capacity](#) is the number of elements that the [ArrayList](#) can store. [Count](#) is the number of elements that are actually in the [ArrayList](#).

[Capacity](#) is always greater than or equal to [Count](#). If [Count](#) exceeds [Capacity](#) while adding elements, the capacity is automatically increased by 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.

See

[Capacity](#)

Also

# ArrayList.FixedSize ArrayList.FixedSize

In this Article

## Overloads

FixedSize(ArrayList) FixedSize(ArrayList)	Returns an <a href="#">ArrayList</a> wrapper with a fixed size.
FixedSize(IList) FixedSize(IList)	Returns an <a href="#">IList</a> wrapper with a fixed size.

## FixedSize(ArrayList) FixedSize(ArrayList)

Returns an [ArrayList](#) wrapper with a fixed size.

```
public static System.Collections.ArrayList FixedSize (System.Collections.ArrayList list);  
static member FixedSize : System.Collections.ArrayList -> System.Collections.ArrayList
```

Parameters

list [ArrayList](#) [ArrayList](#)

The [ArrayList](#) to wrap.

Returns

[ArrayList](#) [ArrayList](#)

An [ArrayList](#) wrapper with a fixed size.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`list` is `null`.

Examples

The following code example shows how to create a fixed-size wrapper around an [ArrayList](#).

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "The" );  
        myAL.Add( "quick" );  
        myAL.Add( "brown" );  
        myAL.Add( "fox" );  
        myAL.Add( "jumps" );  
        myAL.Add( "over" );  
        myAL.Add( "the" );  
        myAL.Add( "lazy" );  
        myAL.Add( "dog" );  
  
        // Create a fixed-size wrapper around the ArrayList.  
        ArrayList myFixedSizeAL = ArrayList.FixedSize( myAL );
```

```

ArrayList myFixedSizeAL = ArrayList.FixedSize( myAL ),

// Display whether the ArrayLists have a fixed size or not.
Console.WriteLine( "myAL {0}.", myAL.IsFixedSize ? "has a fixed size" : "does not have a fixed
size" );
Console.WriteLine( "myFixedSizeAL {0}."., myFixedSizeAL.IsFixedSize ? "has a fixed size" :
"does not have a fixed size" );
Console.WriteLine();

// Display both ArrayLists.
Console.WriteLine( "Initially," );
Console.Write( "Standard :" );
PrintValues( myAL, ' ' );
Console.Write( "Fixed size:" );
PrintValues( myFixedSizeAL, ' ' );

// Sort is allowed in the fixed-size ArrayList.
myFixedSizeAL.Sort();

// Display both ArrayLists.
Console.WriteLine( "After Sort," );
Console.Write( "Standard :" );
PrintValues( myAL, ' ' );
Console.Write( "Fixed size:" );
PrintValues( myFixedSizeAL, ' ' );

// Reverse is allowed in the fixed-size ArrayList.
myFixedSizeAL.Reverse();

// Display both ArrayLists.
Console.WriteLine( "After Reverse," );
Console.Write( "Standard :" );
PrintValues( myAL, ' ' );
Console.Write( "Fixed size:" );
PrintValues( myFixedSizeAL, ' ' );

// Add an element to the standard ArrayList.
myAL.Add( "AddMe" );

// Display both ArrayLists.
Console.WriteLine( "After adding to the standard ArrayList," );
Console.Write( "Standard :" );
PrintValues( myAL, ' ' );
Console.Write( "Fixed size:" );
PrintValues( myFixedSizeAL, ' ' );
Console.WriteLine();

// Adding or inserting elements to the fixed-size ArrayList throws an exception.
try {
    myFixedSizeAL.Add( "AddMe2" );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
try {
    myFixedSizeAL.Insert( 3, "InsertMe" );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

public static void PrintValues( IEnumerable myList, char mySeparator ) {
    foreach ( Object obj in myList )
        Console.Write( "{0}{1}", mySeparator, obj );
    Console.WriteLine();
}

```

```

}

/*
This code produces the following output.

myAL does not have a fixed size.
myFixedSizeAL has a fixed size.

Initially,
Standard : The quick brown fox jumps over the lazy dog
Fixed size: The quick brown fox jumps over the lazy dog
After Sort,
Standard : brown dog fox jumps lazy over quick the The
Fixed size: brown dog fox jumps lazy over quick the The
After Reverse,
Standard : The the quick over lazy jumps fox dog brown
Fixed size: The the quick over lazy jumps fox dog brown
After adding to the standard ArrayList,
Standard : The the quick over lazy jumps fox dog brown AddMe
Fixed size: The the quick over lazy jumps fox dog brown AddMe

Exception: System.NotSupportedException: Collection was of a fixed size.
    at System.Collections.FixedSizeArrayList.Add(Object obj)
    at SamplesArrayList.Main()
Exception: System.NotSupportedException: Collection was of a fixed size.
    at System.Collections.FixedSizeArrayList.Insert(Int32 index, Object obj)
    at SamplesArrayList.Main()

*/

```

## Remarks

This wrapper can be used to prevent additions to and deletions from the original [ArrayList](#). The elements can still be modified or replaced.

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.

This method is an O(1) operation.

## FixedSize(IList) FixedSize(IList)

Returns an [IList](#) wrapper with a fixed size.

```

public static System.Collections.IList FixedSize (System.Collections.IList list);
static member FixedSize : System.Collections.IList -> System.Collections.IList

```

## Parameters

list

[IList](#) [IList](#)

The [IList](#) to wrap.

Returns

[IList](#) [IList](#)

An [IList](#) wrapper with a fixed size.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`list` is `null`.

## Remarks

This wrapper can be used to prevent additions to and deletions from the original [IList](#). The elements can still be modified or replaced.

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.

This method is an O(1) operation.

# ArrayList.GetEnumerator ArrayList.GetEnumerator

In this Article

## Overloads

<a href="#">GetEnumerator()</a> <a href="#">GetEnumerator()</a>	Returns an enumerator for the entire <a href="#">ArrayList</a> .
<a href="#">GetEnumerator(Int32, Int32)</a> <a href="#">GetEnumerator(Int32, Int32)</a>	Returns an enumerator for a range of elements in the <a href="#">ArrayList</a> .

## GetEnumerator() GetEnumerator()

Returns an enumerator for the entire [ArrayList](#).

```
public virtual System.Collections.IEnumerator GetEnumerator ();  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the entire [ArrayList](#).

### Examples

The following example gets the enumerator for an [ArrayList](#), and the enumerator for a range of elements in the [ArrayList](#).

```

using System;
using System.Collections;

class Program
{
    static void Main(string[] args)
    {
        ArrayList colors = new ArrayList();
        colors.Add("red");
        colors.Add("blue");
        colors.Add("green");
        colors.Add("yellow");
        colors.Add("beige");
        colors.Add("brown");
        colors.Add("magenta");
        colors.Add("purple");

        IEnumerator e = colors.GetEnumerator();
        while (e.MoveNext())
        {
            Object obj = e.Current;
            Console.WriteLine(obj);
        }

        Console.WriteLine();

        IEnumerator e2 = colors.GetEnumerator(2, 4);
        while (e2.MoveNext())
        {
            Object obj = e2.Current;
            Console.WriteLine(obj);
        }
    }
}

/* This code example produces
the following output:
red
blue
green
yellow
beige
brown
magenta
purple

green
yellow
beige
brown
*/

```

## 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 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.

This method is an O(1) operation.

See

[IEnumerator](#)[IEnumerator](#)

Also

## GetEnumerator(Int32, Int32) GetEnumerator(Int32, Int32)

Returns an enumerator for a range of elements in the [ArrayList](#).

```
public virtual System.Collections.IEnumerator GetEnumerator (int index, int count);  
  
abstract member GetEnumerator : int * int -> System.Collections.IEnumerator  
override this.GetEnumerator : int * int -> System.Collections.IEnumerator
```

Parameters

index

[Int32](#) [Int32](#)

The zero-based starting index of the [ArrayList](#) section that the enumerator should refer to.

count

[Int32](#) [Int32](#)

The number of elements in the [ArrayList](#) section that the enumerator should refer to.

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the specified range of elements in the [ArrayList](#).

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`count` is less than zero.

[ArgumentException](#) [ArgumentException](#)

`index` and `count` do not specify a valid range in the [ArrayList](#).

Examples

The following example gets the enumerator for an [ArrayList](#), and the enumerator for a range of elements in the [ArrayList](#).

```

using System;
using System.Collections;

class Program
{
    static void Main(string[] args)
    {
        ArrayList colors = new ArrayList();
        colors.Add("red");
        colors.Add("blue");
        colors.Add("green");
        colors.Add("yellow");
        colors.Add("beige");
        colors.Add("brown");
        colors.Add("magenta");
        colors.Add("purple");

        IEnumerator e = colors.GetEnumerator();
        while (e.MoveNext())
        {
            Object obj = e.Current;
            Console.WriteLine(obj);
        }

        Console.WriteLine();

        IEnumerator e2 = colors.GetEnumerator(2, 4);
        while (e2.MoveNext())
        {
            Object obj = e2.Current;
            Console.WriteLine(obj);
        }
    }
}

/* This code example produces
the following output:
red
blue
green
yellow
beige
brown
magenta
purple

green
yellow
beige
brown
*/

```

## Remarks

The `foreach` statement of the C# language (`for each` in Visual C++, `For Each` 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 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.

This method is an O(1) operation.

#### Version Compatibility

In the .NET Framework versions 1.0 and 1.1, the enumerator for an `ArrayList` wrapper returned by the `Adapter` method treated the second argument as an upper bound rather than as a count. In the .NET Framework 2.0 the second argument is correctly treated as a count.

See

[IEnumerator](#)[IEnumerator](#)

Also

# ArrayList.GetRange ArrayList.GetRange

## In this Article

Returns an [ArrayList](#) which represents a subset of the elements in the source [ArrayList](#).

```
public virtual System.Collections.ArrayList GetRange (int index, int count);  
abstract member GetRange : int * int -> System.Collections.ArrayList  
override this.GetRange : int * int -> System.Collections.ArrayList
```

## Parameters

index [Int32](#) [Int32](#)

The zero-based [ArrayList](#) index at which the range starts.

count [Int32](#) [Int32](#)

The number of elements in the range.

## Returns

[ArrayList](#) [ArrayList](#)

An [ArrayList](#) which represents a subset of the elements in the source [ArrayList](#).

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`count` is less than zero.

[ArgumentException](#) [ArgumentException](#)

`index` and `count` do not denote a valid range of elements in the [ArrayList](#).

## Examples

The following code example shows how to set and get a range of elements in the [ArrayList](#).

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Creates and initializes the source ICollection.
        Queue mySourceList = new Queue();
        mySourceList.Enqueue( "big" );
        mySourceList.Enqueue( "gray" );
        mySourceList.Enqueue( "wolf" );

        // Displays the values of five elements starting at index 0.
        ArrayList mySubAL = myAL.GetRange( 0, 5 );
        Console.WriteLine( "Index 0 through 4 contains:" );
        PrintValues( mySubAL, ' ' );

        // Replaces the values of five elements starting at index 1 with the values in the
        // ICollection.
        myAL.SetRange( 1, mySourceList );

        // Displays the values of five elements starting at index 0.
        mySubAL = myAL.GetRange( 0, 5 );
        Console.WriteLine( "Index 0 through 4 now contains:" );
        PrintValues( mySubAL, ' ' );
    }

    public static void PrintValues( IEnumerable myList, char mySeparator ) {
        foreach ( Object obj in myList )
            Console.Write( "{0}{1}", mySeparator, obj );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Index 0 through 4 contains:
    The    quick    brown    fox    jumps
Index 0 through 4 now contains:
    The    big    gray    wolf    jumps
*/

```

## Remarks

This method does not create copies of the elements. The new [ArrayList](#) is only a view window into the source [ArrayList](#). However, all subsequent changes to the source [ArrayList](#) must be done through this view window [ArrayList](#). If changes are made directly to the source [ArrayList](#), the view window [ArrayList](#) is invalidated and any operations on it will return

an [InvalidOperationException](#).

This method is an O(1) operation.

See

[RemoveRange\(Int32, Int32\)](#)

[RemoveRange\(Int32, Int32\)](#)

Also

[AddRange\(IList\)](#)

[AddRange\(IList\)](#)

[InsertRange\(Int32, IList\)](#)

[InsertRange\(Int32, IList\)](#)

[SetRange\(Int32, IList\)](#)

[SetRange\(Int32, IList\)](#)

# ArrayList.IndexOf ArrayList.IndexOf

In this Article

## Overloads

<a href="#">IndexOf(Object) IndexOf(Object)</a>	Searches for the specified <a href="#">Object</a> and returns the zero-based index of the first occurrence within the entire <a href="#">ArrayList</a> .
<a href="#">IndexOf(Object, Int32) IndexOf(Object, Int32)</a>	Searches for the specified <a href="#">Object</a> and returns the zero-based index of the first occurrence within the range of elements in the <a href="#">ArrayList</a> that extends from the specified index to the last element.
<a href="#">IndexOf(Object, Int32, Int32) IndexOf(Object, Int32, Int32)</a>	Searches for the specified <a href="#">Object</a> and returns the zero-based index of the first occurrence within the range of elements in the <a href="#">ArrayList</a> that starts at the specified index and contains the specified number of elements.

## IndexOf(Object) IndexOf(Object)

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the entire [ArrayList](#).

```
public virtual int IndexOf (object value);  
abstract member IndexOf : obj -> int  
override this.IndexOf : obj -> int
```

Parameters

**value** [Object](#) [Object](#)

The [Object](#) to locate in the [ArrayList](#). The value can be [null](#).

Returns

[Int32](#) [Int32](#)

The zero-based index of the first occurrence of [value](#) within the entire [ArrayList](#), if found; otherwise, -1.

Examples

The following code example shows how to determine the index of the first occurrence of a specified element.

```
using System;  
using System.Collections;  
public class SamplesArrayList  
{  
  
    public static void Main()  
{  
  
        // Creates and initializes a new ArrayList with three elements of the same value.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "the" );  
        myAL.Add( "quick" );  
        myAL.Add( "brown" );
```

```

myAL.Add( "fox" );
myAL.Add( "jumps" );
myAL.Add( "over" );
myAL.Add( "the" );
myAL.Add( "lazy" );
myAL.Add( "dog" );
myAL.Add( "in" );
myAL.Add( "the" );
myAL.Add( "barn" );

// Displays the values of the ArrayList.
Console.WriteLine( "The ArrayList contains the following values:" );
PrintIndexAndValues( myAL );

// Search for the first occurrence of the duplicated value.
String myString = "the";
int myIndex = myAL.IndexOf( myString );
Console.WriteLine( "The first occurrence of \"{0}\" is at index {1}.", myString, myIndex );

// Search for the first occurrence of the duplicated value in the last section of the
ArrayList.
myIndex = myAL.IndexOf( myString, 4 );
Console.WriteLine( "The first occurrence of \"{0}\" between index 4 and the end is at index
{1}.", myString, myIndex );

// Search for the first occurrence of the duplicated value in a section of the ArrayList.
myIndex = myAL.IndexOf( myString, 6, 6 );
Console.WriteLine( "The first occurrence of \"{0}\" between index 6 and index 11 is at index
{1}.", myString, myIndex );

// Search for the first occurrence of the duplicated value in a small section at the end of
the ArrayList.
myIndex = myAL.IndexOf( myString, 11 );
Console.WriteLine( "The first occurrence of \"{0}\" between index 11 and the end is at index
{1}.", myString, myIndex );
}

public static void PrintIndexAndValues(IEnumerable myList)
{
    int i = 0;
    foreach (Object obj in myList)
        Console.WriteLine("  [{0}]:  {1}", i++, obj);
    Console.WriteLine();
}

/*
This code produces output similar to the following:

```

The ArrayList contains the following values:

```

[0]:  the
[1]:  quick
[2]:  brown
[3]:  fox
[4]:  jumps
[5]:  over
[6]:  the
[7]:  lazy
[8]:  dog
[9]:  in
[10]: the
[11]: barn

```

The first occurrence of "the" is at index 0.

The first occurrence of "the" between index 4 and the end is at index 6.

```
the first occurrence of "the" between index 6 and index 11 is at index 6.  
The first occurrence of "the" between index 11 and the end is at index -1.  
*/
```

## Remarks

The [ArrayList](#) is searched forward starting at the first element and ending at the last element.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[LastIndexOf\(Object\)](#)

Also

[Contains\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

## IndexOf(Object, Int32) IndexOf(Object, Int32)

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the range of elements in the [ArrayList](#) that extends from the specified index to the last element.

```
public virtual int IndexOf (object value, int startIndex);  
  
abstract member IndexOf : obj * int -> int  
override this.IndexOf : obj * int -> int
```

### Parameters

value

[Object](#)

The [Object](#) to locate in the [ArrayList](#). The value can be [null](#).

startIndex

[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 [value](#) within the range of elements in the [ArrayList](#) that extends from [startIndex](#) to the last element, if found; otherwise, -1.

### Exceptions

[ArgumentOutOfRangeException](#)

[startIndex](#) is outside the range of valid indexes for the [ArrayList](#).

## Examples

The following code example shows how to determine the index of the first occurrence of a specified element.

```
using System;  
using System.Collections;  
public class SamplesArrayList  
{  
  
    public static void Main()  
    {
```

```

// Creates and initializes a new ArrayList with three elements of the same value.
ArrayList myAL = new ArrayList();
myAL.Add( "the" );
myAL.Add( "quick" );
myAL.Add( "brown" );
myAL.Add( "fox" );
myAL.Add( "jumps" );
myAL.Add( "over" );
myAL.Add( "the" );
myAL.Add( "lazy" );
myAL.Add( "dog" );
myAL.Add( "in" );
myAL.Add( "the" );
myAL.Add( "barn" );

// Displays the values of the ArrayList.
Console.WriteLine( "The ArrayList contains the following values:" );
PrintIndexAndValues( myAL );

// Search for the first occurrence of the duplicated value.
String myString = "the";
int myIndex = myAL.IndexOf( myString );
Console.WriteLine( "The first occurrence of \"{0}\" is at index {1}.", myString, myIndex );

// Search for the first occurrence of the duplicated value in the last section of the
ArrayList.
myIndex = myAL.IndexOf( myString, 4 );
Console.WriteLine( "The first occurrence of \"{0}\" between index 4 and the end is at index
{1}.", myString, myIndex );

// Search for the first occurrence of the duplicated value in a section of the ArrayList.
myIndex = myAL.IndexOf( myString, 6, 6 );
Console.WriteLine( "The first occurrence of \"{0}\" between index 6 and index 11 is at index
{1}.", myString, myIndex );

// Search for the first occurrence of the duplicated value in a small section at the end of
the ArrayList.
myIndex = myAL.IndexOf( myString, 11 );
Console.WriteLine( "The first occurrence of \"{0}\" between index 11 and the end is at index
{1}.", myString, myIndex );
}

public static void PrintIndexAndValues(IEnumerable myList)
{
    int i = 0;
    foreach (Object obj in myList)
        Console.WriteLine("  [{0}]:  {1}", i++, obj);
    Console.WriteLine();
}

/*
This code produces output similar to the following:

```

The ArrayList contains the following values:

```

[0]:  the
[1]:  quick
[2]:  brown
[3]:  fox
[4]:  jumps
[5]:  over
[6]:  the
[7]:  lazy
[8]:  dog

```

```
[9]:   in
[10]:   the
[11]:   barn

The first occurrence of "the" is at index 0.
The first occurrence of "the" between index 4 and the end is at index 6.
The first occurrence of "the" between index 6 and index 11 is at index 6.
The first occurrence of "the" between index 11 and the end is at index -1.
*/
```

## Remarks

The [ArrayList](#) is searched forward starting at `startIndex` and ending at the last element.

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 [ArrayList](#).

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

[LastIndexOf\(Object\)](#)

Also

[Contains\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

## IndexOf(Object, Int32, Int32) IndexOf(Object, Int32, Int32)

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the range of elements in the [ArrayList](#) that starts at the specified index and contains the specified number of elements.

```
public virtual int IndexOf (object value, int startIndex, int count);

abstract member IndexOf : obj * int * int -> int
override this.IndexOf : obj * int * int -> int
```

## Parameters

value

[Object](#)

The [Object](#) to locate in the [ArrayList](#). The value can be `null`.

startIndex

[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 `value` within the range of elements in the [ArrayList](#) that starts at `startIndex` and contains `count` number of elements, if found; otherwise, -1.

Exceptions

[ArgumentOutOfRangeException](#)

`startIndex` is outside the range of valid indexes for the [ArrayList](#).

-or-

`count` is less than zero.

-or-

`startIndex` and `count` do not specify a valid section in the [ArrayList](#).

## Examples

The following code example shows how to determine the index of the first occurrence of a specified element.

```
using System;
using System.Collections;
public class SamplesArrayList
{
    public static void Main()
    {
        // Creates and initializes a new ArrayList with three elements of the same value.
        ArrayList myAL = new ArrayList();
        myAL.Add( "the" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );
        myAL.Add( "in" );
        myAL.Add( "the" );
        myAL.Add( "barn" );

        // Displays the values of the ArrayList.
        Console.WriteLine( "The ArrayList contains the following values:" );
        PrintIndexAndValues( myAL );

        // Search for the first occurrence of the duplicated value.
        String myString = "the";
        int myIndex = myAL.IndexOf( myString );
        Console.WriteLine( "The first occurrence of \"{0}\" is at index {1}.", myString, myIndex );

        // Search for the first occurrence of the duplicated value in the last section of the
        // ArrayList.
        myIndex = myAL.IndexOf( myString, 4 );
        Console.WriteLine( "The first occurrence of \"{0}\" between index 4 and the end is at index
        {1}.", myString, myIndex );

        // Search for the first occurrence of the duplicated value in a section of the ArrayList.
        myIndex = myAL.IndexOf( myString, 6, 6 );
        Console.WriteLine( "The first occurrence of \"{0}\" between index 6 and index 11 is at index
        {1}.", myString, myIndex );

        // Search for the first occurrence of the duplicated value in a small section at the end of
        // the ArrayList.
        myIndex = myAL.IndexOf( myString, 11 );
        Console.WriteLine( "The first occurrence of \"{0}\" between index 11 and the end is at index
        {1}.", myString, myIndex );
    }

    public static void PrintIndexAndValues(IEnumerable myList)
```

```
{  
    int i = 0;  
    foreach (Object obj in myList)  
        Console.WriteLine("  [{0}]: {1}", i++, obj);  
    Console.WriteLine();  
}  
}  
/*  
This code produces output similar to the following:
```

The ArrayList contains the following values:

```
[0]: the  
[1]: quick  
[2]: brown  
[3]: fox  
[4]: jumps  
[5]: over  
[6]: the  
[7]: lazy  
[8]: dog  
[9]: in  
[10]: the  
[11]: barn
```

The first occurrence of "the" is at index 0.

The first occurrence of "the" between index 4 and the end is at index 6.

The first occurrence of "the" between index 6 and index 11 is at index 6.

The first occurrence of "the" between index 11 and the end is at index -1.

\*/

## Remarks

The [ArrayList](#) is searched forward starting at `startIndex` and ending at `startIndex` plus `count` minus 1, if `count` is greater than 0.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where `n` is `count`.

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

[LastIndexOf\(Object\)](#)

Also

[Contains\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# ArrayList.Insert ArrayList.Insert

## In this Article

Inserts an element into the [ArrayList](#) at the specified index.

```
public virtual void Insert (int index, object value);  
  
abstract member Insert : int * obj -> unit  
override this.Insert : int * obj -> unit
```

## Parameters

index	<a href="#">Int32</a>
-------	-----------------------

The zero-based index at which `value` should be inserted.

value	<a href="#">Object</a>
-------	------------------------

The [Object](#) to insert. The value can be `null`.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentException](#)

`index` is less than zero.

-or-

`index` is greater than `Count`.

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to insert elements into the [ArrayList](#).

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList using Insert instead of Add.  
        ArrayList myAL = new ArrayList();  
        myAL.Insert( 0, "The" );  
        myAL.Insert( 1, "fox" );  
        myAL.Insert( 2, "jumps" );  
        myAL.Insert( 3, "over" );  
        myAL.Insert( 4, "the" );  
        myAL.Insert( 5, "dog" );  
  
        // Creates and initializes a new Queue.  
        Queue myQueue = new Queue();  
        myQueue.Enqueue( "quick" );  
        myQueue.Enqueue( "brown" );  
  
        // Displays the ArrayList and the Queue.  
        Console.WriteLine("The ArrayList contains the following elements:  
        ");  
        foreach (object item in myAL)  
            Console.WriteLine(item);  
  
        Console.WriteLine("The Queue contains the following elements:  
        ");  
        foreach (object item in myQueue)  
            Console.WriteLine(item);  
    }  
}
```

```

// Displays the ArrayList and the queue.
Console.WriteLine( "The ArrayList initially contains the following:" );
PrintValues( myAL );
Console.WriteLine( "The Queue initially contains the following:" );
PrintValues( myQueue );

// Copies the Queue elements to the ArrayList at index 1.
myAL.InsertRange( 1, myQueue );

// Displays the ArrayList.
Console.WriteLine( "After adding the Queue, the ArrayList now contains:" );
PrintValues( myAL );

// Search for "dog" and add "lazy" before it.
myAL.Insert( myAL.IndexOf( "dog" ), "lazy" );

// Displays the ArrayList.
Console.WriteLine( "After adding \"lazy\", the ArrayList now contains:" );
PrintValues( myAL );

// Add "!!!" at the end.
myAL.Insert( myAL.Count, "!!!" );

// Displays the ArrayList.
Console.WriteLine( "After adding \"!!!\", the ArrayList now contains:" );
PrintValues( myAL );

// Inserting an element beyond Count throws an exception.
try {
    myAL.Insert( myAL.Count+1, "anystring" );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

public static void PrintValues( IEnumerable myList ) {
    foreach ( Object obj in myList )
        Console.Write( "    {0}", obj );
    Console.WriteLine();
}
}
/*
This code produces the following output.

```

The ArrayList initially contains the following:  
 The fox jumps over the dog  
 The Queue initially contains the following:  
 quick brown  
 After adding the Queue, the ArrayList now contains:  
 The quick brown fox jumps over the dog  
 After adding "lazy", the ArrayList now contains:  
 The quick brown fox jumps over the lazy dog  
 After adding "!!!", the ArrayList now contains:  
 The quick brown fox jumps over the lazy dog !!!  
 Exception: System.ArgumentOutOfRangeException: Insertion index was out of range. Must be non-negative and less than or equal to size.  
 Parameter name: index  
 at System.Collections.ArrayList.Insert(Int32 index, Object value)  
 at SamplesArrayList.Main()  
\*/

## Remarks

`ArrayList` accepts `null` as a valid value and allows duplicate elements.

If `Count` already equals `Capacity`, the capacity of the `ArrayList` 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`, `value` is added to the end of `ArrayList`.

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.

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

See

[InsertRange\(Int32, ICollection\)](#)

Also

[Add\(Object\)](#)

[Remove\(Object\)](#)

# ArrayList.InsertRange ArrayList.InsertRange

## In this Article

Inserts the elements of a collection into the [ArrayList](#) at the specified index.

```
public virtual void InsertRange (int index, System.Collections.ICollection c);  
abstract member InsertRange : int * System.Collections.ICollection -> unit  
override this.InsertRange : int * System.Collections.ICollection -> unit
```

## Parameters

index	<a href="#">Int32</a>
-------	-----------------------

The zero-based index at which the new elements should be inserted.

c	<a href="#">ICollection</a>
---	-----------------------------

The [ICollection](#) whose elements should be inserted into the [ArrayList](#). The collection itself cannot be [null](#), but it can contain elements that are [null](#).

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[index](#) is less than zero.

-or-

[index](#) is greater than [Count](#).

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to insert elements into the [ArrayList](#).

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList using Insert instead of Add.  
        ArrayList myAL = new ArrayList();  
        myAL.Insert( 0, "The" );  
        myAL.Insert( 1, "fox" );  
        myAL.Insert( 2, "jumps" );  
        myAL.Insert( 3, "over" );  
        myAL.Insert( 4, "the" );  
        myAL.Insert( 5, "dog" );  
    }  
}
```

```

// Creates and initializes a new Queue.
Queue myQueue = new Queue();
myQueue.Enqueue( "quick" );
myQueue.Enqueue( "brown" );

// Displays the ArrayList and the Queue.
Console.WriteLine( "The ArrayList initially contains the following:" );
PrintValues( myAL );
Console.WriteLine( "The Queue initially contains the following:" );
PrintValues( myQueue );

// Copies the Queue elements to the ArrayList at index 1.
myAL.InsertRange( 1, myQueue );

// Displays the ArrayList.
Console.WriteLine( "After adding the Queue, the ArrayList now contains:" );
PrintValues( myAL );

// Search for "dog" and add "lazy" before it.
myAL.Insert( myAL.IndexOf( "dog" ), "lazy" );

// Displays the ArrayList.
Console.WriteLine( "After adding \"lazy\", the ArrayList now contains:" );
PrintValues( myAL );

// Add "!!!" at the end.
myAL.Insert( myAL.Count, "!!!" );

// Displays the ArrayList.
Console.WriteLine( "After adding \"!!!\", the ArrayList now contains:" );
PrintValues( myAL );

// Inserting an element beyond Count throws an exception.
try {
    myAL.Insert( myAL.Count+1, "anystring" );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

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

/*
This code produces the following output.

The ArrayList initially contains the following:
    The fox jumps over the dog
The Queue initially contains the following:
    quick brown
After adding the Queue, the ArrayList now contains:
    The quick brown fox jumps over the dog
After adding "lazy", the ArrayList now contains:
    The quick brown fox jumps over the lazy dog
After adding "!!!", the ArrayList now contains:
    The quick brown fox jumps over the lazy dog !!!
Exception: System.ArgumentOutOfRangeException: Insertion index was out of range. Must be non-negative and less than or equal to size.
Parameter name: index
    at System.Collections.ArrayList.Insert(Int32 index, Object value)
    at SamplesArrayList.Main()
*/

```

^/

## Remarks

[ArrayList](#) accepts `null` as a valid value 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 [ArrayList](#) 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 [ArrayList](#).

The order of the elements in the [ICollection](#) is preserved in the [ArrayList](#).

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.

This method is an  $O(n + m)$  operation, where `n` is the number of elements to be added and `m` is [Count](#).

See

[Insert\(Int32, Object\)](#)[Insert\(Int32, Object\)](#)

Also

[AddRange\(ICollection\)](#)[AddRange\(ICollection\)](#)

[SetRange\(Int32, ICollection\)](#)[SetRange\(Int32, ICollection\)](#)

[GetRange\(Int32, Int32\)](#)[GetRange\(Int32, Int32\)](#)

[RemoveRange\(Int32, Int32\)](#)[RemoveRange\(Int32, Int32\)](#)

# ArrayList.IsFixedSize ArrayList.IsFixedSize

## In this Article

Gets a value indicating whether the [ArrayList](#) has a fixed size.

```
public virtual bool IsFixedSize { get; }  
member this.IsFixedSize : bool
```

Returns

[Boolean](#)

`true` if the [ArrayList](#) has a fixed size; otherwise, `false`. The default is `false`.

## Examples

The following code example shows how to create a fixed-size wrapper around an [ArrayList](#).

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "The" );  
        myAL.Add( "quick" );  
        myAL.Add( "brown" );  
        myAL.Add( "fox" );  
        myAL.Add( "jumps" );  
        myAL.Add( "over" );  
        myAL.Add( "the" );  
        myAL.Add( "lazy" );  
        myAL.Add( "dog" );  
  
        // Create a fixed-size wrapper around the ArrayList.  
        ArrayList myFixedSizeAL = ArrayList.FixedSize( myAL );  
  
        // Display whether the ArrayLists have a fixed size or not.  
        Console.WriteLine( "myAL {0}."., myAL.IsFixedSize ? "has a fixed size" : "does not have a fixed  
size" );  
        Console.WriteLine( "myFixedSizeAL {0}."., myFixedSizeAL.IsFixedSize ? "has a fixed size" :  
"does not have a fixed size" );  
        Console.WriteLine();  
  
        // Display both ArrayLists.  
        Console.WriteLine( "Initially," );  
        Console.Write( "Standard :" );  
        PrintValues( myAL, ' ' );  
        Console.Write( "Fixed size:" );  
        PrintValues( myFixedSizeAL, ' ' );  
  
        // Sort is allowed in the fixed-size ArrayList.  
        myFixedSizeAL.Sort();  
  
        // Display both ArrayLists.  
        Console.WriteLine( "After Sort," );  
        Console.Write( "Standard :" );  
        PrintValues( myAL, ' ' );  
        Console.Write( "Fixed size:" );  
        PrintValues( myFixedSizeAL, ' ' );
```

```

// Reverse is allowed in the fixed-size ArrayList.
myFixedSizeAL.Reverse();

// Display both ArrayLists.
Console.WriteLine( "After Reverse," );
Console.Write( "Standard :" );
PrintValues( myAL, ' ' );
Console.Write( "Fixed size:" );
PrintValues( myFixedSizeAL, ' ' );

// Add an element to the standard ArrayList.
myAL.Add( "AddMe" );

// Display both ArrayLists.
Console.WriteLine( "After adding to the standard ArrayList," );
Console.Write( "Standard :" );
PrintValues( myAL, ' ' );
Console.Write( "Fixed size:" );
PrintValues( myFixedSizeAL, ' ' );
Console.WriteLine();

// Adding or inserting elements to the fixed-size ArrayList throws an exception.
try {
    myFixedSizeAL.Add( "AddMe2" );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
try {
    myFixedSizeAL.Insert( 3, "InsertMe" );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

public static void PrintValues( IEnumerable myList, char mySeparator ) {
    foreach ( Object obj in myList )
        Console.Write( "{0}{1}", mySeparator, obj );
    Console.WriteLine();
}
}
/*
This code produces the following output.

```

myAL does not have a fixed size.  
myFixedSizeAL has a fixed size.

Initially,  
Standard : The quick brown fox jumps over the lazy dog  
Fixed size: The quick brown fox jumps over the lazy dog  
After Sort,  
Standard : brown dog fox jumps lazy over quick the The  
Fixed size: brown dog fox jumps lazy over quick the The  
After Reverse,  
Standard : The the quick over lazy jumps fox dog brown  
Fixed size: The the quick over lazy jumps fox dog brown  
After adding to the standard ArrayList,  
Standard : The the quick over lazy jumps fox dog brown AddMe  
Fixed size: The the quick over lazy jumps fox dog brown AddMe

Exception: System.NotSupportedException: Collection was of a fixed size.  
at System.CollectionsFixedSizeArrayList.Add(Object obj)  
at SamplesArrayList.Main()  
Exception: System.NotSupportedException: Collection was of a fixed size.  
at System.CollectionsFixedSizeArrayList.Insert(Int32 index, Object obj)

```
at System.Collections.FixedSizeArrayList.Insert(Int32 index, Object obj)
at SamplesArrayList.Main()
```

```
*/
```

## 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.

# ArrayList.IsReadOnly ArrayList.IsReadOnly

## In this Article

Gets a value indicating whether the [ArrayList](#) is read-only.

```
public virtual bool IsReadOnly { get; }  
member this.IsReadOnly : bool
```

Returns

[Boolean](#)

`true` if the [ArrayList](#) is read-only; otherwise, `false`. The default is `false`.

## Examples

The following code example shows how to create a read-only wrapper around an [ArrayList](#) and how to determine if an [ArrayList](#) is read-only.

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "red" );  
        myAL.Add( "orange" );  
        myAL.Add( "yellow" );  
  
        // Creates a read-only copy of the ArrayList.  
        ArrayList myReadOnlyAL = ArrayList.ReadOnly( myAL );  
  
        // Displays whether the ArrayList is read-only or writable.  
        Console.WriteLine( "myAL is {0}.", myAL.IsReadOnly ? "read-only" : "writable" );  
        Console.WriteLine( "myReadOnlyAL is {0}.", myReadOnlyAL.IsReadOnly ? "read-only" : "writable" );  
    }  
  
    // Displays the contents of both collections.  
    Console.WriteLine( "  
Initially," );  
    Console.WriteLine( "The original ArrayList myAL contains:" );  
    foreach ( String myStr in myAL )  
        Console.WriteLine( "    {0}", myStr );  
    Console.WriteLine( "The read-only ArrayList myReadOnlyAL contains:" );  
    foreach ( String myStr in myReadOnlyAL )  
        Console.WriteLine( "    {0}", myStr );  
  
    // Adding an element to a read-only ArrayList throws an exception.  
    Console.WriteLine( "  
Trying to add a new element to the read-only ArrayList:" );  
    try {  
        myReadOnlyAL.Add("green");  
    } catch ( Exception myException ) {  
        Console.WriteLine("Exception: " + myException.ToString());  
    }  
  
    // Adding an element to the original ArrayList affects the read-only ArrayList.  
    myAL.Add( "blue" );  
  
    // Displays the contents of both collections again.
```

```

        Console.WriteLine( "After adding a new element to the original ArrayList," );
        Console.WriteLine( "The original ArrayList myAL contains:" );
        foreach ( String myStr in myAL )
            Console.WriteLine( "    {0}", myStr );
        Console.WriteLine( "The read-only ArrayList myReadOnlyAL contains:" );
        foreach ( String myStr in myReadOnlyAL )
            Console.WriteLine( "    {0}", myStr );

    }

}

/*
This code produces the following output.

myAL is writable.
myReadOnlyAL is read-only.

Initially,
The original ArrayList myAL contains:
    red
    orange
    yellow
The read-only ArrayList myReadOnlyAL contains:
    red
    orange
    yellow

Trying to add a new element to the read-only ArrayList:
Exception: System.NotSupportedException: Collection is read-only.
    at System.Collections.ReadOnlyArrayList.Add(Object obj)
    at SamplesArrayList.Main()

After adding a new element to the original ArrayList,
The original ArrayList myAL contains:
    red
    orange
    yellow
    blue
The read-only ArrayList myReadOnlyAL contains:
    red
    orange
    yellow
    blue

*/

```

## 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.

See

[ReadOnly\(IList\)](#)[ReadOnly\(IList\)](#)

Also

# ArrayList.IsSynchronized ArrayList.IsSynchronized

## In this Article

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

```
public virtual bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean](#)

`true` if access to the [ArrayList](#) is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
ArrayList myCollection = new ArrayList();  
  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

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

The following code example shows how to synchronize an [ArrayList](#), determine if an [ArrayList](#) is synchronized and use a synchronized [ArrayList](#).

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );

        // Creates a synchronized wrapper around the ArrayList.
        ArrayList mySyncdAL = ArrayList.Synchronized( myAL );

        // Displays the synchronization status of both ArrayLists.
        Console.WriteLine( "myAL is {0}.", myAL.IsSynchronized ? "synchronized" : "not synchronized" );
        Console.WriteLine( "mySyncdAL is {0}.", mySyncdAL.IsSynchronized ? "synchronized" : "not synchronized" );
    }
}

/*
This code produces the following output.

myAL is not synchronized.
mySyncdAL is synchronized.
*/

```

## Remarks

To guarantee the thread safety of the [ArrayList](#), all operations must be done through the wrapper returned by the [Synchronized](#) method.

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 causes 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.

See

[SyncRoot](#)

Also

[Synchronized\(IList\)](#)

# ArrayList.Item[Int32] ArrayList.Item[Int32]

## In this Article

Gets or sets the element at the specified index.

```
public virtual object this[int index] { get; set; }  
member this.Item(int) : obj with get, set
```

## Parameters

index Int32 Int32

The zero-based index of the element to get or set.

## Returns

[Object Object](#)

The element at the specified index.

## Exceptions

[ArgumentOutOfRangeException ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` is equal to or greater than `Count`.

## Examples

The following code example creates an [ArrayList](#) and adds several items. The example demonstrates accessing elements with the [Item\[Int32\]](#) property (the indexer in C#), and changing an element by assigning a new value to the [Item\[Int32\]](#) property for a specified index. The example also shows that the [Item\[Int32\]](#) property cannot be used to access or add elements outside the current size of the list.

```
using System;  
using System.Collections;  
  
public class Example  
{  
    public static void Main()  
    {  
        // Create an empty ArrayList, and add some elements.  
        ArrayList stringList = new ArrayList();  
  
        stringList.Add("a");  
        stringList.Add("abc");  
        stringList.Add("abcdef");  
        stringList.Add("abcdefg");  
  
        // The Item property is an indexer, so the property name is  
        // not required.  
        Console.WriteLine("Element {0} is \"{1}\", 2, stringList[2]);  
  
        // Assigning a value to the property changes the value of  
        // the indexed element.  
        stringList[2] = "abcd";  
        Console.WriteLine("Element {0} is \"{1}\", 2, stringList[2]);  
  
        // Accessing an element outside the current element count
```

```

// Accessing an element outside the current element count
// causes an exception.
Console.WriteLine("Number of elements in the list: {0}",
    stringList.Count);
try
{
    Console.WriteLine("Element {0} is \"{1}\",
        stringList.Count, stringList[stringList.Count]);
}
catch(ArgumentOutOfRangeException aoore)
{
    Console.WriteLine("stringList({0}) is out of range.",
        stringList.Count);
}

// You cannot use the Item property to add new elements.
try
{
    stringList[stringList.Count] = "42";
}
catch(ArgumentOutOfRangeException aoore)
{
    Console.WriteLine("stringList({0}) is out of range.",
        stringList.Count);
}

Console.WriteLine();
for (int i = 0; i < stringList.Count; i++)
{
    Console.WriteLine("Element {0} is \"{1}\",
        i, stringList[i]);
}

Console.WriteLine();
foreach (object o in stringList)
{
    Console.WriteLine(o);
}
}
*/

```

This code example produces the following output:

```

Element 2 is "abcdef"
Element 2 is "abcd"
Number of elements in the list: 4
stringList(4) is out of range.
stringList(4) is out of range.

Element 0 is "a"
Element 1 is "abc"
Element 2 is "abcd"
Element 3 is "abcdefg"

a
abc
abcd
abcdefg
*/

```

The following example uses the [Item\[Int32\]](#) property explicitly to assign values to items in the list. The example defines a class that inherits an [ArrayList](#) and adds a method to scramble the list items.

```
using System;
using System.Collections;

public class ScrambleList : ArrayList
{
    public static void Main()
    {
        // Create an empty ArrayList, and add some elements.
        ScrambleList integerList = new ScrambleList();

        for (int i = 0; i < 10; i++)
        {
            integerList.Add(i);
        }

        Console.WriteLine("Ordered:");
    );
        foreach (int value in integerList)
        {
            Console.Write("{0}, ", value);
        }
        Console.WriteLine("<end>

Scrambled:
");
        // Scramble the order of the items in the list.
        integerList.Scramble();

        foreach (int value in integerList)
        {
            Console.Write("{0}, ", value);
        }
        Console.WriteLine("<end>
");
    }
}

public void Scramble()
{
    int limit = this.Count;
    int temp;
    int swapindex;
    Random rnd = new Random();
    for (int i = 0; i < limit; i++)
    {
        // The Item property of ArrayList is the default indexer. Thus,
        // this[i] is used instead of Item[i].
        temp = (int)this[i];
        swapindex = rnd.Next(0, limit - 1);
        this[i] = this[swapindex];
        this[swapindex] = temp;
    }
}

// The program produces output similar to the following:
// 
// Ordered:
// 
// 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, <end>
// 
// Scrambled:
// 
// 5, 2, 8, 9, 6, 1, 7, 0, 4, 3, <end>
```

## Remarks

The `Item[Int32]` returns an `Object`, so you may need to cast the returned value to the original type in order to manipulate it. It is important to note that `ArrayList` is not a strongly-typed collection. For a strongly-typed alternative, see [List<T>](#).

`ArrayList` accepts `null` as a valid value and allows duplicate elements.

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 keyword to define the indexers instead of implementing the `Item[Int32]` property. Visual Basic implements `Item[Int32]` 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.

See

[Count](#)

Also

# ArrayList.LastIndexOf ArrayList.LastIndexOf

In this Article

## Overloads

<code>LastIndexOf(Object) LastIndexOf(Object)</code>	Searches for the specified <a href="#">Object</a> and returns the zero-based index of the last occurrence within the entire <a href="#">ArrayList</a> .
<code>LastIndexOf(Object, Int32) LastIndexOf(Object, Int32)</code>	Searches for the specified <a href="#">Object</a> and returns the zero-based index of the last occurrence within the range of elements in the <a href="#">ArrayList</a> that extends from the first element to the specified index.
<code>LastIndexOf(Object, Int32, Int32) LastIndexOf(Object, Int32, Int32)</code>	Searches for the specified <a href="#">Object</a> and returns the zero-based index of the last occurrence within the range of elements in the <a href="#">ArrayList</a> that contains the specified number of elements and ends at the specified index.

## LastIndexOf(Object) LastIndexOf(Object)

Searches for the specified [Object](#) and returns the zero-based index of the last occurrence within the entire [ArrayList](#).

```
public virtual int LastIndexOf (object value);  
abstract member LastIndexOf : obj -> int  
override this.LastIndexOf : obj -> int
```

Parameters

`value` [Object](#) [Object](#)

The [Object](#) to locate in the [ArrayList](#). The value can be `null`.

Returns

[Int32](#) [Int32](#)

The zero-based index of the last occurrence of `value` within the entire the [ArrayList](#), if found; otherwise, -1.

Examples

The following code example shows how to determine the index of the last occurrence of a specified element.

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList with three elements of the same value.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "the" );  
        myAL.Add( "quick" );  
        myAL.Add( "brown" );  
        myAL.Add( "fox" );  
        myAL.Add( "jumps" );
```

```

myAL.Add( "over" );
myAL.Add( "the" );
myAL.Add( "lazy" );
myAL.Add( "dog" );
myAL.Add( "in" );
myAL.Add( "the" );
myAL.Add( "barn" );

// Displays the values of the ArrayList.
Console.WriteLine( "The ArrayList contains the following values:" );
PrintIndexAndValues( myAL );

// Searches for the last occurrence of the duplicated value.
String myString = "the";
int myIndex = myAL.LastIndexOf( myString );
Console.WriteLine( "The last occurrence of \"{0}\" is at index {1}.", myString, myIndex );

// Searches for the last occurrence of the duplicated value in the first section of the
ArrayList.
myIndex = myAL.LastIndexOf( myString, 8 );
Console.WriteLine( "The last occurrence of \"{0}\" between the start and index 8 is at index
{1}.", myString, myIndex );

// Searches for the last occurrence of the duplicated value in a section of the ArrayList.
Note that the start index is greater than the end index because the search is done backward.
myIndex = myAL.LastIndexOf( myString, 10, 6 );
Console.WriteLine( "The last occurrence of \"{0}\" between index 10 and index 5 is at index
{1}.", myString, myIndex );
}

public static void PrintIndexAndValues( IEnumerable myList ) {
    int i = 0;
    foreach ( Object obj in myList )
        Console.WriteLine( "[{0}]: {1}", i++, obj );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

The ArrayList contains the following values:

```

[0]: the
[1]: quick
[2]: brown
[3]: fox
[4]: jumps
[5]: over
[6]: the
[7]: lazy
[8]: dog
[9]: in
[10]: the
[11]: barn

```

The last occurrence of "the" is at index 10.

The last occurrence of "the" between the start and index 8 is at index 6.

The last occurrence of "the" between index 10 and index 5 is at index 10.

\*/

## Remarks

The [ArrayList](#) is searched backward starting at the last element and ending at the first element.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

#### **IndexOf(Object)IndexOf(Object)**

Also

**Contains(Object)Contains(Object)**

## Performing Culture-Insensitive String Operations in Collections

**LastIndexOf(Object, Int32)** **LastIndexOf(Object, Int32)**

Searches for the specified [Object](#) and returns the zero-based index of the last occurrence within the range of elements in the [ArrayList](#) that extends from the first element to the specified index.

```
public virtual int LastIndexOf (object value, int startIndex);  
  
abstract member LastIndexOf : obj * int -> int  
override this.LastIndexOf : obj * int -> int
```

## Parameters

value

## Object Object

The `Object` to locate in the `ArrayList`. The value can be `null`.

startIndex

Int32 Int32

The zero-based starting index of the backward search.

## Returns

Int32 Int32

The zero-based index of the last occurrence of `value` within the range of elements in the `ArrayList` that extends from the first element to `startIndex`, if found; otherwise, -1.

## Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`startIndex` is outside the range of valid indexes for the `ArrayList`.

## Examples

The following code example shows how to determine the index of the last occurrence of a specified element.

```
using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList with three elements of the same value.
        ArrayList myAL = new ArrayList();
        myAL.Add( "the" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
    }
}
```

```

myAL.Add( "dog" );
myAL.Add( "in" );
myAL.Add( "the" );
myAL.Add( "barn" );

// Displays the values of the ArrayList.
Console.WriteLine( "The ArrayList contains the following values:" );
PrintIndexAndValues( myAL );

// Searches for the last occurrence of the duplicated value.
String myString = "the";
int myIndex = myAL.LastIndexOf( myString );
Console.WriteLine( "The last occurrence of \"{0}\" is at index {1}.", myString, myIndex );

// Searches for the last occurrence of the duplicated value in the first section of the
ArrayList.
myIndex = myAL.LastIndexOf( myString, 8 );
Console.WriteLine( "The last occurrence of \"{0}\" between the start and index 8 is at index
{1}.", myString, myIndex );

// Searches for the last occurrence of the duplicated value in a section of the ArrayList.
Note that the start index is greater than the end index because the search is done backward.
myIndex = myAL.LastIndexOf( myString, 10, 6 );
Console.WriteLine( "The last occurrence of \"{0}\" between index 10 and index 5 is at index
{1}.", myString, myIndex );
}

public static void PrintIndexAndValues( IEnumerable myList ) {
    int i = 0;
    foreach ( Object obj in myList )
        Console.WriteLine( "[{0}]: {1}", i++, obj );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

The ArrayList contains the following values:

```

[0]: the
[1]: quick
[2]: brown
[3]: fox
[4]: jumps
[5]: over
[6]: the
[7]: lazy
[8]: dog
[9]: in
[10]: the
[11]: barn

```

The last occurrence of "the" is at index 10.

The last occurrence of "the" between the start and index 8 is at index 6.

The last occurrence of "the" between index 10 and index 5 is at index 10.

\*/

## Remarks

The [ArrayList](#) is searched backward starting at `startIndex` and ending at the first element.

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 [ArrayList](#) to `startIndex`.

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[IndexOf\(Object\)](#)[IndexOf\(Object\)](#)

Also

[Contains\(Object\)](#)[Contains\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

## LastIndexOf(Object, Int32, Int32) LastIndexOf(Object, Int32, Int32)

Searches for the specified [Object](#) and returns the zero-based index of the last occurrence within the range of elements in the [ArrayList](#) that contains the specified number of elements and ends at the specified index.

```
public virtual int LastIndexOf (object value, int startIndex, int count);  
abstract member LastIndexOf : obj * int * int -> int  
override this.LastIndexOf : obj * int * int -> int
```

Parameters

value

[Object](#) [Object](#)

The [Object](#) to locate in the [ArrayList](#). The value can be [null](#).

startIndex

[Int32](#) [Int32](#)

The zero-based starting index of the backward search.

count

[Int32](#) [Int32](#)

The number of elements in the section to search.

Returns

[Int32](#) [Int32](#)

The zero-based index of the last occurrence of [value](#) within the range of elements in the [ArrayList](#) that contains [count](#) number of elements and ends at [startIndex](#), if found; otherwise, -1.

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[startIndex](#) is outside the range of valid indexes for the [ArrayList](#).

-or-

[count](#) is less than zero.

-or-

[startIndex](#) and [count](#) do not specify a valid section in the [ArrayList](#).

Examples

The following code example shows how to determine the index of the last occurrence of a specified element. Note that [LastIndexOf](#) is a backward search; therefore, [count](#) must be less than or equal to [startIndex](#) + 1.

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

```

public static void Main() {

    // Creates and initializes a new ArrayList with three elements of the same value.
    ArrayList myAL = new ArrayList();
    myAL.Add( "the" );
    myAL.Add( "quick" );
    myAL.Add( "brown" );
    myAL.Add( "fox" );
    myAL.Add( "jumps" );
    myAL.Add( "over" );
    myAL.Add( "the" );
    myAL.Add( "lazy" );
    myAL.Add( "dog" );
    myAL.Add( "in" );
    myAL.Add( "the" );
    myAL.Add( "barn" );

    // Displays the values of the ArrayList.
    Console.WriteLine( "The ArrayList contains the following values:" );
    PrintIndexAndValues( myAL );

    // Searches for the last occurrence of the duplicated value.
    String myString = "the";
    int myIndex = myAL.LastIndexOf( myString );
    Console.WriteLine( "The last occurrence of \"{0}\" is at index {1}.", myString, myIndex );

    // Searches for the last occurrence of the duplicated value in the first section of the
    // ArrayList.
    myIndex = myAL.LastIndexOf( myString, 8 );
    Console.WriteLine( "The last occurrence of \"{0}\" between the start and index 8 is at index
{1}.", myString, myIndex );

    // Searches for the last occurrence of the duplicated value in a section of the ArrayList.
    Note that the start index is greater than the end index because the search is done backward.
    myIndex = myAL.LastIndexOf( myString, 10, 6 );
    Console.WriteLine( "The last occurrence of \"{0}\" between index 10 and index 5 is at index
{1}.", myString, myIndex );
}

public static void PrintIndexAndValues( IEnumerable myList ) {
    int i = 0;
    foreach ( Object obj in myList )
        Console.WriteLine( "[{0}]: {1}", i++, obj );
    Console.WriteLine();
}

/*
This code produces the following output.

The ArrayList contains the following values:
[0]: the
[1]: quick
[2]: brown
[3]: fox
[4]: jumps
[5]: over
[6]: the
[7]: lazy
[8]: dog
[9]: in
[10]: the

```

```
[11]: barn
```

```
The last occurrence of "the" is at index 10.  
The last occurrence of "the" between the start and index 8 is at index 6.  
The last occurrence of "the" between index 10 and index 5 is at index 10.  
*/
```

## Remarks

The [ArrayList](#) is searched backward starting at `startIndex` and ending at `startIndex` minus `count` plus 1, if `count` is greater than 0.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where `n` is `count`.

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

[IndexOf\(Object\)](#)[IndexOf\(Object\)](#)

Also

[Contains\(Object\)](#)[Contains\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# ArrayList.ReadOnly ArrayList.ReadOnly

In this Article

## Overloads

<a href="#">ReadOnly(ArrayList)</a> <a href="#">ReadOnly(ArrayList)</a>	Returns a read-only <a href="#">ArrayList</a> wrapper.
<a href="#">ReadOnly(IList)</a> <a href="#">ReadOnly(IList)</a>	Returns a read-only <a href="#">IList</a> wrapper.

## ReadOnly(ArrayList) ReadOnly(ArrayList)

Returns a read-only [ArrayList](#) wrapper.

```
public static System.Collections.ArrayList ReadOnly (System.Collections.ArrayList list);  
static member ReadOnly : System.Collections.ArrayList -> System.Collections.ArrayList
```

Parameters

list [ArrayList](#) [ArrayList](#)

The [ArrayList](#) to wrap.

Returns

[ArrayList](#) [ArrayList](#)

A read-only [ArrayList](#) wrapper around `list`.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`list` is `null`.

Examples

The following code example shows how to create a read-only wrapper around an [ArrayList](#) and how to determine if an [ArrayList](#) is read-only.

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "red" );  
        myAL.Add( "orange" );  
        myAL.Add( "yellow" );  
  
        // Creates a read-only copy of the ArrayList.  
        ArrayList myReadOnlyAL = ArrayList.ReadOnly( myAL );  
  
        // Displays whether the ArrayList is read-only or writable.  
        Console.WriteLine( "myAL is {0}.". , myAL.IsReadOnly ? "read-only" : "writable" );  
        Console.WriteLine( "myReadOnlyAL is {0}.". , myReadOnlyAL.IsReadOnly ? "read-only" : "writable" );  
    }  
}
```

```

);

// Displays the contents of both collections.
Console.WriteLine( "
Initially," );
Console.WriteLine( "The original ArrayList myAL contains:" );
foreach ( String myStr in myAL )
    Console.WriteLine( "    {0}", myStr );
Console.WriteLine( "The read-only ArrayList myReadOnlyAL contains:" );
foreach ( String myStr in myReadOnlyAL )
    Console.WriteLine( "    {0}", myStr );

// Adding an element to a read-only ArrayList throws an exception.
Console.WriteLine( "
Trying to add a new element to the read-only ArrayList:" );
try {
    myReadOnlyAL.Add("green");
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}

// Adding an element to the original ArrayList affects the read-only ArrayList.
myAL.Add( "blue" );

// Displays the contents of both collections again.
Console.WriteLine( "
After adding a new element to the original ArrayList," );
Console.WriteLine( "The original ArrayList myAL contains:" );
foreach ( String myStr in myAL )
    Console.WriteLine( "    {0}", myStr );
Console.WriteLine( "The read-only ArrayList myReadOnlyAL contains:" );
foreach ( String myStr in myReadOnlyAL )
    Console.WriteLine( "    {0}", myStr );

}
}

/*
This code produces the following output.

```

myAL is writable.  
myReadOnlyAL is read-only.

Initially,  
The original ArrayList myAL contains:  
red  
orange  
yellow  
The read-only ArrayList myReadOnlyAL contains:  
red  
orange  
yellow

Trying to add a new element to the read-only ArrayList:  
Exception: System.NotSupportedException: Collection is read-only.  
at System.Collections.ReadOnlyArrayList.Add(Object obj)  
at SamplesArrayList.Main()

After adding a new element to the original ArrayList,  
The original ArrayList myAL contains:  
red  
orange  
yellow  
blue

```
blue
The read-only ArrayList myReadOnlyAL contains:
red
orange
yellow
blue

*/
```

## Remarks

To prevent any modifications to `list`, expose `list` only through this wrapper.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection. If changes are made to the underlying collection, the read-only collection reflects those changes.

This method is an O(1) operation.

See

[IsReadOnly](#)[IsReadOnly](#)

Also

## ReadOnly(IList) ReadOnly(IList)

Returns a read-only [IList](#) wrapper.

```
public static System.Collections.IList ReadOnly (System.Collections.IList list);
static member ReadOnly : System.Collections.IList -> System.Collections.IList
```

## Parameters

list

[IList](#)[IList](#)

The [IList](#) to wrap.

Returns

[IList](#)[IList](#)

A read-only [IList](#) wrapper around `list`.

Exceptions

[ArgumentNullException](#)[ArgumentNullException](#)

`list` is `null`.

## Remarks

To prevent any modifications to `list`, expose `list` only through this wrapper.

A collection that is read-only is simply a collection with a wrapper that prevents modifying the collection. If changes are made to the underlying collection, the read-only collection reflects those changes.

This method is an O(1) operation.

See

[IsReadOnly](#)[IsReadOnly](#)

Also

# ArrayList.Remove ArrayList.Remove

## In this Article

Removes the first occurrence of a specific object from the [ArrayList](#).

```
public virtual void Remove (object obj);  
  
abstract member Remove : obj -> unit  
override this.Remove : obj -> unit
```

## Parameters

obj	<a href="#">Object</a>
-----	------------------------

The [Object](#) to remove from the [ArrayList](#). The value can be `null`.

## Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to remove elements from the [ArrayList](#).

```

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

    public static void Main() {

        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Displays the ArrayList.
        Console.WriteLine( "The ArrayList initially contains the following:" );
        PrintValues( myAL );

        // Removes the element containing "lazy".
        myAL.Remove( "lazy" );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing \"lazy\":" );
        PrintValues( myAL );

        // Removes the element at index 5.
        myAL.RemoveAt( 5 );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing the element at index 5:" );
        PrintValues( myAL );

        // Removes three elements starting at index 4.
        myAL.RemoveRange( 4, 3 );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing three elements starting at index 4:" );
        PrintValues( myAL );
    }

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

/*
This code produces the following output.

The ArrayList initially contains the following:
    The    quick    brown    fox    jumps    over    the    lazy    dog
After removing "lazy":
    The    quick    brown    fox    jumps    over    the    dog
After removing the element at index 5:
    The    quick    brown    fox    jumps    the    dog
After removing three elements starting at index 4:
    The    quick    brown    fox
*/

```

## Remarks

If the [ArrayList](#) does not contain the specified object, the [ArrayList](#) remains unchanged. No exception is thrown.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

This method determines equality by calling [Object.Equals](#).

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.

See

[RemoveAt\(Int32\)](#)[RemoveAt\(Int32\)](#)

Also

[RemoveRange\(Int32, Int32\)](#)[RemoveRange\(Int32, Int32\)](#)

[Add\(Object\)](#)[Add\(Object\)](#)

[Insert\(Int32, Object\)](#)[Insert\(Int32, Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# ArrayList.RemoveAt ArrayList.RemoveAt

## In this Article

Removes the element at the specified index of the [ArrayList](#).

```
public virtual void RemoveAt (int index);  
  
abstract member RemoveAt : int -> unit  
override this.RemoveAt : int -> unit
```

## Parameters

index	<a href="#">Int32</a>	<a href="#">Int32</a>
-------	-----------------------	-----------------------

The zero-based index of the element to remove.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` is equal to or greater than `Count`.

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to remove elements from the [ArrayList](#).

```

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

    public static void Main() {

        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Displays the ArrayList.
        Console.WriteLine( "The ArrayList initially contains the following:" );
        PrintValues( myAL );

        // Removes the element containing "lazy".
        myAL.Remove( "lazy" );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing \"lazy\":" );
        PrintValues( myAL );

        // Removes the element at index 5.
        myAL.RemoveAt( 5 );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing the element at index 5:" );
        PrintValues( myAL );

        // Removes three elements starting at index 4.
        myAL.RemoveRange( 4, 3 );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing three elements starting at index 4:" );
        PrintValues( myAL );
    }

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

/*
This code produces the following output.

The ArrayList initially contains the following:
    The    quick    brown    fox    jumps    over    the    lazy    dog
After removing "lazy":
    The    quick    brown    fox    jumps    over    the    dog
After removing the element at index 5:
    The    quick    brown    fox    jumps    the    dog
After removing three elements starting at index 4:
    The    quick    brown    fox
*/

```

## Remarks

After the element is removed, the size of the collection is adjusted and the value of the [Count](#) property is decreased by one.

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.

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

See

[Remove\(Object\)](#)  
[Remove\(Object\)](#)

Also

[RemoveRange\(Int32, Int32\)](#)  
[RemoveRange\(Int32, Int32\)](#)  
[Add\(Object\)](#)  
[Add\(Object\)](#)  
[Insert\(Int32, Object\)](#)  
[Insert\(Int32, Object\)](#)

# ArrayList.RemoveRange ArrayList.RemoveRange

## In this Article

Removes a range of elements from the [ArrayList](#).

```
public virtual void RemoveRange (int index, int count);  
  
abstract member RemoveRange : int * int -> unit  
override this.RemoveRange : int * int -> unit
```

## Parameters

index	<a href="#">Int32</a>	<a href="#">Int32</a>
-------	-----------------------	-----------------------

The zero-based starting index of the range of elements to remove.

count	<a href="#">Int32</a>	<a href="#">Int32</a>
-------	-----------------------	-----------------------

The number of elements to remove.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentException](#)

`index` is less than zero.

-or-

`count` is less than zero.

[ArgumentException](#) [ArgumentException](#)

`index` and `count` do not denote a valid range of elements in the [ArrayList](#).

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to remove elements from the [ArrayList](#).

```

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

    public static void Main() {

        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Displays the ArrayList.
        Console.WriteLine( "The ArrayList initially contains the following:" );
        PrintValues( myAL );

        // Removes the element containing "lazy".
        myAL.Remove( "lazy" );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing \"lazy\":" );
        PrintValues( myAL );

        // Removes the element at index 5.
        myAL.RemoveAt( 5 );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing the element at index 5:" );
        PrintValues( myAL );

        // Removes three elements starting at index 4.
        myAL.RemoveRange( 4, 3 );

        // Displays the current state of the ArrayList.
        Console.WriteLine( "After removing three elements starting at index 4:" );
        PrintValues( myAL );
    }

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

/*
This code produces the following output.

The ArrayList initially contains the following:
    The    quick    brown    fox    jumps    over    the    lazy    dog
After removing "lazy":
    The    quick    brown    fox    jumps    over    the    dog
After removing the element at index 5:
    The    quick    brown    fox    jumps    the    dog
After removing three elements starting at index 4:
    The    quick    brown    fox
*/

```

## 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.

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

See

[Remove\(Object\)](#)  
[RemoveAt\(Int32\)](#)

Also

[GetRange\(Int32, Int32\)](#)  
[AddRange\(ICollection\)](#)  
[InsertRange\(Int32, ICollection\)](#)  
[SetRange\(Int32, ICollection\)](#)

# ArrayList.Repeat ArrayList.Repeat

## In this Article

Returns an [ArrayList](#) whose elements are copies of the specified value.

```
public static System.Collections.ArrayList Repeat (object value, int count);  
static member Repeat : obj * int -> System.Collections.ArrayList
```

## Parameters

value Object Object

The [Object](#) to copy multiple times in the new [ArrayList](#). The value can be `null`.

count Int32 Int32

The number of times `value` should be copied.

## Returns

[ArrayList](#) [ArrayList](#)

An [ArrayList](#) with `count` number of elements, all of which are copies of `value`.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`count` is less than zero.

## Examples

The following code example shows how to create and initialize a new [ArrayList](#) with the same value.

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates a new ArrayList with five elements and initialize each element with a null value.
        ArrayList myAL = ArrayList.Repeat( null, 5 );
        // Displays the count, capacity and values of the ArrayList.
        Console.WriteLine( "ArrayList with five elements with a null value" );
        Console.WriteLine( "    Count      : {0}", myAL.Count );
        Console.WriteLine( "    Capacity   : {0}", myAL.Capacity );
        Console.Write( "    Values:" );
        PrintValues( myAL );
        // Creates a new ArrayList with seven elements and initialize each element with the string
        // "abc".
        myAL = ArrayList.Repeat( "abc", 7 );
        // Displays the count, capacity and values of the ArrayList.
        Console.WriteLine( "ArrayList with seven elements with a string value" );
        Console.WriteLine( "    Count      : {0}", myAL.Count );
        Console.WriteLine( "    Capacity   : {0}", myAL.Capacity );
        Console.Write( "    Values:" );
        PrintValues( myAL );
    }
    public static void PrintValues( IEnumerable myList ) {
        foreach ( Object obj in myList )
            Console.Write( "    {0}", obj );
        Console.WriteLine();
    }
}
/*
This code produces the following output.

ArrayList with five elements with a null value
    Count      : 5
    Capacity   : 16
    Values:
ArrayList with seven elements with a string value
    Count      : 7
    Capacity   : 16
    Values:    abc    abc    abc    abc    abc    abc    abc
*/

```

## Remarks

[ArrayList](#) accepts `null` as a valid value and allows duplicate elements.

This method is an  $O(n)$  operation, where `n` is `count`.

# ArrayList.Reverse ArrayList.Reverse

In this Article

## Overloads

<a href="#">Reverse()</a> <a href="#">Reverse()</a>	Reverses the order of the elements in the entire <a href="#">ArrayList</a> .
<a href="#">Reverse(Int32, Int32)</a> <a href="#">Reverse(Int32, Int32)</a>	Reverses the order of the elements in the specified range.

### Reverse() Reverse()

Reverses the order of the elements in the entire [ArrayList](#).

```
public virtual void Reverse ();  
  
abstract member Reverse : unit -> unit  
override this.Reverse : unit -> unit
```

Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

Examples

The following code example shows how to reverse the sort order of the values in an [ArrayList](#).

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Displays the values of the ArrayList.
        Console.WriteLine( "The ArrayList initially contains the following values:" );
        PrintValues( myAL );

        // Reverses the sort order of the values of the ArrayList.
        myAL.Reverse();

        // Displays the values of the ArrayList.
        Console.WriteLine( "After reversing:" );
        PrintValues( myAL );
    }

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

/*
This code produces the following output.

The ArrayList initially contains the following values:
The
quick
brown
fox
jumps
over
the
lazy
dog

After reversing:
dog
lazy
the
over
jumps
fox
brown
quick
The
*/

```

## Remarks

This method uses [ArrayList.Reverse](#) to reverse the order of the elements, such that the element at [ArrayList\[i\]](#), where i is any index within the range, moves to [ArrayList\[j\]](#), where j equals [index + index + count - i - 1](#).

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

## Reverse(Int32, Int32) Reverse(Int32, Int32)

Reverses the order of the elements in the specified range.

```
public virtual void Reverse (int index, int count);  
  
abstract member Reverse : int * int -> unit  
override this.Reverse : int * int -> unit
```

### Parameters

index [Int32](#) [Int32](#)

The zero-based starting index of the range to reverse.

count [Int32](#) [Int32](#)

The number of elements in the range to reverse.

### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[index](#) is less than zero.

-or-

[count](#) is less than zero.

[ArgumentException](#) [ArgumentException](#)

[index](#) and [count](#) do not denote a valid range of elements in the [ArrayList](#).

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

### Examples

The following code example shows how to reverse the sort order of the values in a range of elements in an [ArrayList](#).

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "The" );  
        myAL.Add( "QUICK" );  
        myAL.Add( "BROWN" );  
        myAL.Add( "FOX" );  
        myAL.Add( "jumps" );  
        myAL.Add( "over" );  
        myAL.Add( "the" );  
        myAL.Add( "lazy" );
```

```

myAL.Add( "dog" );

// Displays the values of the ArrayList.
Console.WriteLine( "The ArrayList initially contains the following values:" );
PrintValues( myAL );

// Reverses the sort order of the values of the ArrayList.
myAL.Reverse( 1, 3 );

// Displays the values of the ArrayList.
Console.WriteLine( "After reversing:" );
PrintValues( myAL );
}

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

/*
This code produces the following output.

The ArrayList initially contains the following values:
The
QUICK
BROWN
FOX
jumps
over
the
lazy
dog

After reversing:
The
FOX
BROWN
QUICK
jumps
over
the
lazy
dog

*/

```

## Remarks

This method uses [Array.Reverse](#) to reverse the order of the elements, such that the element at [ArrayList\[i\]](#), where i is any index within the range, moves to [ArrayList\[j\]](#), where j equals [index + index + count - i - 1](#).

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

# ArrayList.SetRange ArrayList.SetRange

## In this Article

Copies the elements of a collection over a range of elements in the [ArrayList](#).

```
public virtual void SetRange (int index, System.Collections.ICollection c);  
abstract member SetRange : int * System.Collections.ICollection -> unit  
override this.SetRange : int * System.Collections.ICollection -> unit
```

## Parameters

index Int32 Int32

The zero-based [ArrayList](#) index at which to start copying the elements of `c`.

c [ICollection](#) [ICollection](#)

The [ICollection](#) whose elements to copy to the [ArrayList](#). The collection itself cannot be `null`, but it can contain elements that are `null`.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` plus the number of elements in `c` is greater than `Count`.

[ArgumentNullException](#) [ArgumentNullException](#)

`c` is `null`.

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

## Examples

The following code example shows how to set and get a range of elements in the [ArrayList](#).

```

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

    public static void Main() {

        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Creates and initializes the source ICollection.
        Queue mySourceList = new Queue();
        mySourceList.Enqueue( "big" );
        mySourceList.Enqueue( "gray" );
        mySourceList.Enqueue( "wolf" );

        // Displays the values of five elements starting at index 0.
        ArrayList mySubAL = myAL.GetRange( 0, 5 );
        Console.WriteLine( "Index 0 through 4 contains:" );
        PrintValues( mySubAL, ' ' );

        // Replaces the values of five elements starting at index 1 with the values in the
        // ICollection.
        myAL.SetRange( 1, mySourceList );

        // Displays the values of five elements starting at index 0.
        mySubAL = myAL.GetRange( 0, 5 );
        Console.WriteLine( "Index 0 through 4 now contains:" );
        PrintValues( mySubAL, ' ' );
    }

    public static void PrintValues( IEnumerable myList, char mySeparator ) {
        foreach ( Object obj in myList )
            Console.Write( "{0}{1}", mySeparator, obj );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Index 0 through 4 contains:
    The    quick    brown    fox    jumps
Index 0 through 4 now contains:
    The    big    gray    wolf    jumps
*/

```

## Remarks

[ArrayList](#) accepts `null` as a valid value and allows duplicate elements.

The order of the elements in the [ICollection](#) is preserved in the [ArrayList](#).

This method is an  $O(n + 1)$  operation, where  $n$  is [Count](#).

See

[AddRange\(IICollection\)](#)[AddRange\(IICollection\)](#)

Also

[InsertRange\(Int32, IICollection\)](#)[InsertRange\(Int32, IICollection\)](#)

[GetRange\(Int32, Int32\)](#)[GetRange\(Int32, Int32\)](#)

[RemoveRange\(Int32, Int32\)](#)[RemoveRange\(Int32, Int32\)](#)

# ArrayList.Sort ArrayList.Sort

In this Article

## Overloads

Sort() Sort()	Sorts the elements in the entire <a href="#">ArrayList</a> .
Sort(IComparer) Sort(IComparer)	Sorts the elements in the entire <a href="#">ArrayList</a> using the specified comparer.
Sort(Int32, Int32, IComparer) Sort(Int32, Int32, IComparer)	Sorts the elements in a range of elements in <a href="#">ArrayList</a> using the specified comparer.

## Sort() Sort()

Sorts the elements in the entire [ArrayList](#).

```
public virtual void Sort ();  
  
abstract member Sort : unit -> unit  
override this.Sort : unit -> unit
```

Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

Examples

The following code example shows how to sort the values in an [ArrayList](#).

```

using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Displays the values of the ArrayList.
        Console.WriteLine( "The ArrayList initially contains the following values:" );
        PrintValues( myAL );

        // Sorts the values of the ArrayList.
        myAL.Sort();

        // Displays the values of the ArrayList.
        Console.WriteLine( "After sorting:" );
        PrintValues( myAL );
    }

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

```

/\*
This code produces the following output.

The ArrayList initially contains the following values:

```

The
quick
brown
fox
jumps
over
the
lazy
dog

```

After sorting:

```

brown
dog
fox
jumps
lazy
over
quick
the
The

```

\*/

## Remarks

This method uses [Array.Sort](#), which uses the QuickSort algorithm. The QuickSort algorithm is a comparison sort (also called an unstable sort), which means that a "less than or equal to" comparison operation determines which of two elements should occur first in the final sorted list. However, if two elements are equal, their original order might not be preserved. In contrast, a stable sort preserves the order of elements that are equal. To perform a stable sort, you must implement a custom [IComparer](#) interface to use with the other overloads of this method.

On average, this method is an  $O(n \log n)$  operation, where  $n$  is [Count](#); in the worst case it is an  $O(n^2)$  operation.

See

[Performing Culture-Insensitive String Operations in Collections](#)

Also

## Sort(IComparer) Sort(IComparer)

Sorts the elements in the entire [ArrayList](#) using the specified comparer.

```
public virtual void Sort (System.Collections.IComparer comparer);  
abstract member Sort : System.Collections.IComparer -> unit  
override this.Sort : System.Collections.IComparer -> unit
```

Parameters

comparer

[IComparer](#)

The [IComparer](#) implementation to use when comparing elements.

-or-

A null reference ([Nothing](#) in Visual Basic) to use the [IComparable](#) implementation of each element.

Exceptions

[NotSupportedException](#)

The [ArrayList](#) is read-only.

[InvalidOperationException](#)

An error occurred while comparing two elements.

[ArgumentException](#)

`null` is passed for `comparer`, and the elements in the list do not implement [IComparable](#).

## Examples

The following code example shows how to sort the values in an [ArrayList](#) using the default comparer and a custom comparer that reverses the sort order.

```
using System;  
using System.Collections;  
  
public class SamplesArrayList {  
  
    public class myReverserClass : IComparer {  
  
        // Calls CaseInsensitiveComparer.Compare with the parameters reversed.  
        int IComparer.Compare( Object x, Object y ) {  
            return( (new CaseInsensitiveComparer()).Compare( y, x ) );  
        }  
    }  
}
```

```

}

public static void Main()  {

    // Creates and initializes a new ArrayList.
    ArrayList myAL = new ArrayList();
    myAL.Add( "The" );
    myAL.Add( "quick" );
    myAL.Add( "brown" );
    myAL.Add( "fox" );
    myAL.Add( "jumps" );
    myAL.Add( "over" );
    myAL.Add( "the" );
    myAL.Add( "lazy" );
    myAL.Add( "dog" );

    // Displays the values of the ArrayList.
    Console.WriteLine( "The ArrayList initially contains the following values:" );
    PrintIndexAndValues( myAL );

    // Sorts the values of the ArrayList using the default comparer.
    myAL.Sort();
    Console.WriteLine( "After sorting with the default comparer:" );
    PrintIndexAndValues( myAL );

    // Sorts the values of the ArrayList using the reverse case-insensitive comparer.
    IComparer myComparer = new myReverserClass();
    myAL.Sort( myComparer );
    Console.WriteLine( "After sorting with the reverse case-insensitive comparer:" );
    PrintIndexAndValues( myAL );

}

public static void PrintIndexAndValues( IEnumerable myList )  {
    int i = 0;
    foreach ( Object obj in myList )
        Console.WriteLine( "    [{0}]: {1}", i++, obj );
    Console.WriteLine();
}

}

/*
This code produces the following output.
The ArrayList initially contains the following values:
    [0]: The
    [1]: quick
    [2]: brown
    [3]: fox
    [4]: jumps
    [5]: over
    [6]: the
    [7]: lazy
    [8]: dog

After sorting with the default comparer:
    [0]: brown
    [1]: dog
    [2]: fox
    [3]: jumps
    [4]: lazy
    [5]: over
    [6]: quick
    [7]: the
    [8]: the

```

```
[8]:    The
After sorting with the reverse case-insensitive comparer:
[0]:    the
[1]:    The
[2]:    quick
[3]:    over
[4]:    lazy
[5]:    jumps
[6]:    fox
[7]:    dog
[8]:    brown
*/
```

## Remarks

Use the [Sort](#) method to sort a list of objects with a custom comparer that implements the [IComparer](#) interface. If you pass `null` for `comparer`, this method uses the [IComparable](#) implementation of each element. In this case, you must make sure that the objects contained in the list implement the [IComparer](#) interface or an exception will occur.

In addition, using the [IComparable](#) implementation means the list performs a comparison sort (also called 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. To perform a stable sort, you must implement a custom [IComparer](#) interface.

On average, this method is an  $O(n \log n)$  operation, where `n` is [Count](#); in the worst case it is an  $O(n^2)$  operation.

See

[Performing Culture-Insensitive String Operations in Collections](#)

Also

## Sort(Int32, Int32, IComparer) Sort(Int32, Int32, IComparer)

Sorts the elements in a range of elements in [ArrayList](#) using the specified comparer.

```
public virtual void Sort (int index, int count, System.Collections.IComparer comparer);
abstract member Sort : int * int * System.Collections.IComparer -> unit
override this.Sort : int * int * System.Collections.IComparer -> unit
```

### Parameters

index	<a href="#">Int32</a>
-------	-----------------------

The zero-based starting index of the range to sort.

count	<a href="#">Int32</a>
-------	-----------------------

The length of the range to sort.

comparer	<a href="#">IComparer</a>
----------	---------------------------

The [IComparer](#) implementation to use when comparing elements.

-or-

A null reference (`Nothing` in Visual Basic) to use the [IComparable](#) implementation of each element.

### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`count` is less than zero.

### ArgumentException ArgumentException

`index` and `count` do not specify a valid range in the [ArrayList](#).

### NotSupportedException NotSupportedException

The [ArrayList](#) is read-only.

### InvalidOperationException InvalidOperationException

An error occurred while comparing two elements.

## Examples

The following code example shows how to sort the values in a range of elements in an [ArrayList](#) using the default comparer and a custom comparer that reverses the sort order.

```
using System;
using System.Collections;

public class SamplesArrayList {

    public class myReverserClass : IComparer {
        // Calls CaseInsensitiveComparer.Compare with the parameters reversed.
        int IComparer.Compare( Object x, Object y ) {
            return( (new CaseInsensitiveComparer()).Compare( y, x ) );
        }
    }

    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "QUICK" );
        myAL.Add( "BROWN" );
        myAL.Add( "FOX" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Displays the values of the ArrayList.
        Console.WriteLine( "The ArrayList initially contains the following values:" );
        PrintIndexAndValues( myAL );

        // Sorts the values of the ArrayList using the default comparer.
        myAL.Sort( 1, 3, null );
        Console.WriteLine( "After sorting from index 1 to index 3 with the default comparer:" );
        PrintIndexAndValues( myAL );

        // Sorts the values of the ArrayList using the reverse case-insensitive comparer.
        IComparer myComparer = new myReverserClass();
        myAL.Sort( 1, 3, myComparer );
        Console.WriteLine( "After sorting from index 1 to index 3 with the reverse case-insensitive comparer:" );
        PrintIndexAndValues( myAL );
    }
}
```

```

}

public static void PrintIndexAndValues( IEnumerable myList ) {
    int i = 0;
    foreach ( Object obj in myList )
        Console.WriteLine( "    [{0}]: {1}", i++, obj );
    Console.WriteLine();
}

/*
This code produces the following output.
The ArrayList initially contains the following values:
[0]: The
[1]: QUICK
[2]: BROWN
[3]: FOX
[4]: jumps
[5]: over
[6]: the
[7]: lazy
[8]: dog

After sorting from index 1 to index 3 with the default comparer:
[0]: The
[1]: BROWN
[2]: FOX
[3]: QUICK
[4]: jumps
[5]: over
[6]: the
[7]: lazy
[8]: dog

After sorting from index 1 to index 3 with the reverse case-insensitive comparer:
[0]: The
[1]: QUICK
[2]: FOX
[3]: BROWN
[4]: jumps
[5]: over
[6]: the
[7]: lazy
[8]: dog
*/

```

## Remarks

If `comparer` is set to `null`, this method performs a comparison sort (also called 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. To perform a stable sort, you must implement a custom [IComparer](#) interface.

On average, this method is an  $O(n \log n)$  operation, where  $n$  is `count`; in the worst case it is an  $O(n^2)$  operation.

See

[Performing Culture-Insensitive String Operations in Collections](#)

Also

# ArrayList.Synchronized ArrayList.Synchronized

In this Article

## Overloads

Synchronized(ArrayList) Synchronized(ArrayList)	Returns an <a href="#">ArrayList</a> wrapper that is synchronized (thread safe).
Synchronized(IList) Synchronized(IList)	Returns an <a href="#">IList</a> wrapper that is synchronized (thread safe).

## Synchronized(ArrayList) Synchronized(ArrayList)

Returns an [ArrayList](#) wrapper that is synchronized (thread safe).

```
public static System.Collections.ArrayList Synchronized (System.Collections.ArrayList list);  
static member Synchronized : System.Collections.ArrayList -> System.Collections.ArrayList
```

Parameters

list [ArrayList](#) [ArrayList](#)

The [ArrayList](#) to synchronize.

Returns

[ArrayList](#) [ArrayList](#)

An [ArrayList](#) wrapper that is synchronized (thread safe).

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`list` is `null`.

Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
ArrayList myCollection = new ArrayList();  
  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

This method is an O(1) operation.

The following code example shows how to synchronize an [ArrayList](#), determine if an [ArrayList](#) is synchronized and use a synchronized [ArrayList](#).

```

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

    public static void Main() {

        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );

        // Creates a synchronized wrapper around the ArrayList.
        ArrayList mySyncdAL = ArrayList.Synchronized( myAL );

        // Displays the synchronization status of both ArrayLists.
        Console.WriteLine( "myAL is {0}."., myAL.IsSynchronized ? "synchronized" : "not synchronized"
);
        Console.WriteLine( "mySyncdAL is {0}."., mySyncdAL.IsSynchronized ? "synchronized" : "not
synchronized" );
    }
}
/*
This code produces the following output.

myAL is not synchronized.
mySyncdAL is synchronized.
*/

```

## Remarks

To guarantee the thread safety of the [ArrayList](#), all operations must be done through this wrapper.

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 causes 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.

### See

[SyncRoot](#)

### Also

[IsSynchronized](#)

[IsSynchronized](#)

## Synchronized(IList) Synchronized(IList)

Returns an [IList](#) wrapper that is synchronized (thread safe).

```

public static System.Collections.IList Synchronized (System.Collections.IList list);
static member Synchronized : System.Collections.IList -> System.Collections.IList

```

### Parameters

#### list

[IList](#)

The [IList](#) to synchronize.

### Returns

[IList](#)

An [IList](#) wrapper that is synchronized (thread safe).

### Exceptions

## ArgumentNullException ArgumentNullException

list is null.

### Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
ArrayList myCollection = new ArrayList();

lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

This method is an O(1) operation.

### Remarks

To guarantee the thread safety of the [ArrayList](#), all operations must be done through this wrapper.

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 causes 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.

See

[SyncRoot](#)

Also

[IsSynchronized](#)

# ArrayList.SyncRoot ArrayList.SyncRoot

## In this Article

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

```
public virtual object SyncRoot { get; }  
member this.SyncRoot : obj
```

Returns

[Object Object](#)

An object that can be used to synchronize access to the [ArrayList](#).

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
ArrayList myCollection = new ArrayList();  
  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

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

## Remarks

To create a synchronized version of the [ArrayList](#), use the [Synchronized](#) method. However, derived classes can provide their own synchronized version of the [ArrayList](#) using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) of the [ArrayList](#), not directly on the [ArrayList](#). This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [ArrayList](#) object.

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 causes 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.

See

[IsSynchronized](#)  
[IsSynchronized](#)

Also

[Synchronized\(IList\)](#)  
[Synchronized\(IList\)](#)

# ArrayList.ToArray ArrayList.ToArray

In this Article

## Overloads

<a href="#">ToArray()</a> <a href="#">ToArray()</a>	Copies the elements of the <a href="#">ArrayList</a> to a new <a href="#">Object</a> array.
<a href="#">ToArray(Type)</a> <a href="#">ToArray(Type)</a>	Copies the elements of the <a href="#">ArrayList</a> to a new array of the specified element type.

### ToArray() ToArray()

Copies the elements of the [ArrayList](#) to a new [Object](#) array.

```
public virtual object[] ToArray ();  
abstract member ToArray : unit -> obj[]  
override this.ToArray : unit -> obj[]
```

Returns

[Object](#)[]

An [Object](#) array containing copies of the elements of the [ArrayList](#).

Remarks

The elements are copied using [Array.Copy](#), which is an O( $n$ ) operation, where  $n$  is [Count](#).

### ToArray(Type) ToArray(Type)

Copies the elements of the [ArrayList](#) to a new array of the specified element type.

```
public virtual Array ToArray (Type type);  
abstract member ToArray : Type -> Array  
override this.ToArray : Type -> Array
```

Parameters

type [Type](#) [Type](#)

The element [Type](#) of the destination array to create and copy elements to.

Returns

[Array](#) [Array](#)

An array of the specified element type containing copies of the elements of the [ArrayList](#).

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`type` is `null`.

[InvalidOperationException](#) [InvalidOperationException](#)

The type of the source [ArrayList](#) cannot be cast automatically to the specified type.

## Examples

The following copy example shows how to copy the elements of an [ArrayList](#) to a string array.

```
using System;
using System.Collections;

public class SamplesArrayList {

    public static void Main() {

        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "The" );
        myAL.Add( "quick" );
        myAL.Add( "brown" );
        myAL.Add( "fox" );
        myAL.Add( "jumps" );
        myAL.Add( "over" );
        myAL.Add( "the" );
        myAL.Add( "lazy" );
        myAL.Add( "dog" );

        // Displays the values of the ArrayList.
        Console.WriteLine( "The ArrayList contains the following values:" );
        PrintIndexAndValues( myAL );

        // Copies the elements of the ArrayList to a string array.
        String[] myArr = (String[]) myAL.ToArray( typeof( string ) );

        // Displays the contents of the string array.
        Console.WriteLine( "The string array contains the following values:" );
        PrintIndexAndValues( myArr );

    }

    public static void PrintIndexAndValues( ArrayList myList ) {
        int i = 0;
        foreach ( Object o in myList )
            Console.WriteLine( "    [{0}]:    {1}", i++, o );
        Console.WriteLine();
    }

    public static void PrintIndexAndValues( String[] myArr ) {
        for ( int i = 0; i < myArr.Length; i++ )
            Console.WriteLine( "    [{0}]:    {1}", i, myArr[i] );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The ArrayList contains the following values:
[0]:    The
[1]:    quick
[2]:    brown
[3]:    fox
[4]:    jumps
[5]:    over
[6]:    the
[7]:    lazy

```

```
[8]:    dog
```

The string array contains the following values:

```
[0]:    The
[1]:    quick
[2]:    brown
[3]:    fox
[4]:    jumps
[5]:    over
[6]:    the
[7]:    lazy
[8]:    dog
```

```
*/
```

## Remarks

All of the objects in the [ArrayList](#) object will be cast to the [Type](#) specified in the `type` parameter.

The elements are copied using [Array.Copy](#), which is an  $O(n)$  operation, where `n` is [Count](#).

See

[Type](#)

Also

# ArrayList.TrimToSize ArrayList.TrimToSize

## In this Article

Sets the capacity to the actual number of elements in the [ArrayList](#).

```
public virtual void TrimToSize ();  
  
abstract member TrimToSize : unit -> unit  
override this.TrimToSize : unit -> unit
```

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [ArrayList](#) is read-only.

-or-

The [ArrayList](#) has a fixed size.

## Examples

The following code example shows how to trim the unused portions of the [ArrayList](#) and how to clear the values of the [ArrayList](#).

```
using System;  
using System.Collections;  
public class SamplesArrayList {  
  
    public static void Main() {  
  
        // Creates and initializes a new ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "The" );  
        myAL.Add( "quick" );  
        myAL.Add( "brown" );  
        myAL.Add( "fox" );  
        myAL.Add( "jumps" );  
  
        // Displays the count, capacity and values of the ArrayList.  
        Console.WriteLine( "Initially," );  
        Console.WriteLine( " Count : {0}", myAL.Count );  
        Console.WriteLine( " Capacity : {0}", myAL.Capacity );  
        Console.Write( " Values:" );  
        PrintValues( myAL );  
  
        // Trim the ArrayList.  
        myAL.TrimToSize();  
  
        // Displays the count, capacity and values of the ArrayList.  
        Console.WriteLine( "After TrimToSize," );  
        Console.WriteLine( " Count : {0}", myAL.Count );  
        Console.WriteLine( " Capacity : {0}", myAL.Capacity );  
        Console.Write( " Values:" );  
        PrintValues( myAL );  
  
        // Clear the ArrayList.  
        myAL.Clear();  
  
        // Displays the count, capacity and values of the ArrayList.  
        Console.WriteLine( "After Clear," );  
        Console.WriteLine( " Count : {0}", myAL.Count );  
        Console.WriteLine( " Capacity : {0}", myAL.Capacity );
```

```

        Console.Write( "    Values:" );
        PrintValues( myAL );

        // Trim the ArrayList again.
        myAL.TrimToSize();

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

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

/*
This code produces the following output.

Initially,
Count      : 5
Capacity   : 16
Values:    The    quick    brown    fox    jumps
After TrimToSize,
Count      : 5
Capacity   : 5
Values:    The    quick    brown    fox    jumps
After Clear,
Count      : 0
Capacity   : 5
Values:
After the second TrimToSize,
Count      : 0
Capacity   : 16
Values:
*/

```

## Remarks

This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection.

To reset a [ArrayList](#) to its initial state, call the [Clear](#) method before calling [TrimToSize](#). Trimming an empty [ArrayList](#) sets the capacity of the [ArrayList](#) to the default capacity.

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

See

[Clear\(\)](#)[Clear\(\)](#)

Also

[Capacity](#)[Capacity](#)

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

# BitArray BitArray Class

Manages a compact array of bit values, which are represented as Booleans, where `true` indicates that the bit is on (1) and `false` indicates the bit is off (0).

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public sealed class BitArray : ICloneable, System.Collections.ICollection

type BitArray = class
    interface ICollection
    interface ICloneable
    interface IEnumerable
```

## Inheritance Hierarchy

`Object` `Object`

## Remarks

The `BitArray` class is a collection class in which the capacity is always the same as the count. Elements are added to a `BitArray` by increasing the `Length` property; elements are deleted by decreasing the `Length` property. The size of a `BitArray` is controlled by the client; indexing past the end of the `BitArray` throws an `ArgumentException`. The `BitArray` class provides methods that are not found in other collections, including those that allow multiple elements to be modified at once using a filter, such as `And`, `Or`, `Xor`, `Not`, and `SetAll`.

The `BitVector32` class is a structure that provides the same functionality as `BitArray`, but with faster performance. `BitVector32` is faster because it is a value type and therefore allocated on the stack, whereas `BitArray` is a reference type and, therefore, allocated on the heap.

`System.Collections.Specialized.BitVector32` can store exactly 32 bits, whereas `BitArray` can store a variable number of bits. `BitVector32` stores both bit flags and small integers, thereby making it ideal for data that is not exposed to the user. However, if the number of required bit flags is unknown, is variable, or is greater than 32, use `BitArray` instead.

`BitArray` is in the `System.Collections` namespace; `BitVector32` is in the `System.Collections.Specialized` namespace.

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

## Constructors

```
BitArray(Boolean[])
BitArray(Boolean[])
```

Initializes a new instance of the `BitArray` class that contains bit values copied from the specified array of Booleans.

```
BitArray(Byte[])
BitArray(Byte[])
```

Initializes a new instance of the `BitArray` class that contains bit values copied from the specified array of bytes.

```
BitArray(BitArray)
BitArray(BitArray)
```

Initializes a new instance of the [BitArray](#) class that contains bit values copied from the specified [BitArray](#).

`BitArray(Int32)`

`BitArray(Int32)`

Initializes a new instance of the [BitArray](#) class that can hold the specified number of bit values, which are initially set to `false`.

`BitArray(Int32[])`

`BitArray(Int32[])`

Initializes a new instance of the [BitArray](#) class that contains bit values copied from the specified array of 32-bit integers.

`BitArray(Int32, Boolean)`

`BitArray(Int32, Boolean)`

Initializes a new instance of the [BitArray](#) class that can hold the specified number of bit values, which are initially set to the specified value.

## Properties

`Count`

`Count`

Gets the number of elements contained in the [BitArray](#).

`IsReadOnly`

`IsReadOnly`

Gets a value indicating whether the [BitArray](#) is read-only.

`IsSynchronized`

`IsSynchronized`

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

`Item[Int32]`

`Item[Int32]`

Gets or sets the value of the bit at a specific position in the [BitArray](#).

`Length`

`Length`

Gets or sets the number of elements in the [BitArray](#).

`SyncRoot`

`SyncRoot`

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

## Methods

`And(BitArray)`

`And(BitArray)`

Performs the bitwise AND operation between the elements of the current [BitArray](#) object and the corresponding elements in the specified array. The current [BitArray](#) object will be modified to store the result of the bitwise AND operation.

`Clone()`

`Clone()`

Creates a shallow copy of the [BitArray](#).

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

Copies the entire [BitArray](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

`Get(Int32)`

`Get(Int32)`

Gets the value of the bit at a specific position in the [BitArray](#).

`GetEnumerator()`

`GetEnumerator()`

Returns an enumerator that iterates through the [BitArray](#).

`LeftShift(Int32)`

`LeftShift(Int32)`

`Not()`

`Not()`

Inverts all the bit values in the current [BitArray](#), so that elements set to `true` are changed to `false`, and elements

set to `false` are changed to `true`.

```
Or(BitArray)
Or(BitArray)
```

Performs the bitwise OR operation between the elements of the current [BitArray](#) object and the corresponding elements in the specified array. The current [BitArray](#) object will be modified to store the result of the bitwise OR operation.

```
RightShift(Int32)
RightShift(Int32)
```

```
Set(Int32, Boolean)
Set(Int32, Boolean)
```

Sets the bit at a specific position in the [BitArray](#) to the specified value.

```
SetAll(Boolean)
SetAll(Boolean)
```

Sets all bits in the [BitArray](#) to the specified value.

```
Xor(BitArray)
Xor(BitArray)
```

Performs the bitwise exclusive OR operation between the elements of the current [BitArray](#) object against the corresponding elements in the specified array. The current [BitArray](#) object will be modified to store the result of the bitwise exclusive OR operation.

```
ICollection.CopyTo(Array, Int32)
ICollection.CopyTo(Array, Int32)
```

```
ICollection.Count
ICollection.Count
```

```
ICollection.IsSynchronized
ICollection.IsSynchronized
```

```
ICollection.SyncRoot
ICollection.SyncRoot
```

## 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.

This implementation does not provide a synchronized (thread safe) wrapper for a [BitArray](#).

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 causes 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.

# BitArray.And BitArray.And

## In this Article

Performs the bitwise AND operation between the elements of the current [BitArray](#) object and the corresponding elements in the specified array. The current [BitArray](#) object will be modified to store the result of the bitwise AND operation.

```
public System.Collections.BitArray And (System.Collections.BitArray value);  
member this.And : System.Collections.BitArray -> System.Collections.BitArray
```

### Parameters

value [BitArray](#) [BitArray](#)

The array with which to perform the bitwise AND operation.

### Returns

[BitArray](#) [BitArray](#)

An array containing the result of the bitwise AND operation, which is a reference to the current [BitArray](#) object.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`value` is `null`.

[ArgumentException](#) [ArgumentException](#)

`value` and the current [BitArray](#) do not have the same number of elements.

## Examples

The following code example shows how to perform the bitwise AND operation between two [BitArray](#) objects.

```
using System;  
using System.Collections;  
public class SamplesBitArray {  
  
    public static void Main() {  
  
        // Creates and initializes two BitArrays of the same size.  
        BitArray myBA1 = new BitArray( 4 );  
        BitArray myBA2 = new BitArray( 4 );  
        myBA1[0] = myBA1[1] = false;  
        myBA1[2] = myBA1[3] = true;  
        myBA2[0] = myBA2[2] = false;  
        myBA2[1] = myBA2[3] = true;  
  
        // Performs a bitwise AND operation between BitArray instances of the same size.  
        Console.WriteLine( "Initial values" );  
        Console.Write( "myBA1:" );  
        PrintValues( myBA1, 8 );  
        Console.Write( "myBA2:" );  
        PrintValues( myBA2, 8 );  
        Console.WriteLine();  
  
        Console.WriteLine( "Result" );  
        Console.Write( "AND:" );  
        PrintValues( myBA1.And( myBA2 ), 8 );  
        Console.WriteLine();
```

```

Console.WriteLine( "After AND" );
Console.Write( "myBA1:" );
PrintValues( myBA1, 8 );
Console.Write( "myBA2:" );
PrintValues( myBA2, 8 );
Console.WriteLine();

// Performing AND between BitArray instances of different sizes returns an exception.
try {
    BitArray myBA3 = new BitArray( 8 );
    myBA3[0] = myBA3[1] = myBA3[2] = myBA3[3] = false;
    myBA3[4] = myBA3[5] = myBA3[6] = myBA3[7] = true;
    myBA1.And( myBA3 );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

public static void PrintValues( IEnumerable myList, int myWidth ) {
    int i = myWidth;
    foreach ( Object obj in myList ) {
        if ( i <= 0 ) {
            i = myWidth;
            Console.WriteLine();
        }
        i--;
        Console.Write( "{0,8}", obj );
    }
    Console.WriteLine();
}

}

/*
This code produces the following output.

Initial values
myBA1:  False  False  True  True
myBA2:  False  True  False  True

Result
AND:  False  False  False  True

After AND
myBA1:  False  False  False  True
myBA2:  False  True  False  True

Exception: System.ArgumentException: Array lengths must be the same.
  at System.Collections.BitArray.And(BitArray value)
  at SamplesBitArray.Main()
*/

```

## Remarks

The bitwise AND operation returns `true` if both operands are `true`, and returns `false` if one or both operands are `false`.

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

# BitArray BitArray

In this Article

## Overloads

<code>BitArray(Boolean[]) BitArray(Boolean[])</code>	Initializes a new instance of the <a href="#">BitArray</a> class that contains bit values copied from the specified array of Booleans.
<code>BitArray(Byte[]) BitArray(Byte[])</code>	Initializes a new instance of the <a href="#">BitArray</a> class that contains bit values copied from the specified array of bytes.
<code>BitArray(BitArray) BitArray(BitArray)</code>	Initializes a new instance of the <a href="#">BitArray</a> class that contains bit values copied from the specified <a href="#">BitArray</a> .
<code>BitArray(Int32) BitArray(Int32)</code>	Initializes a new instance of the <a href="#">BitArray</a> class that can hold the specified number of bit values, which are initially set to <code>false</code> .
<code>BitArray(Int32[]) BitArray(Int32[])</code>	Initializes a new instance of the <a href="#">BitArray</a> class that contains bit values copied from the specified array of 32-bit integers.
<code>BitArray(Int32, Boolean) BitArray(Int32, Boolean)</code>	Initializes a new instance of the <a href="#">BitArray</a> class that can hold the specified number of bit values, which are initially set to the specified value.

## BitArray(Boolean[]) BitArray(Boolean[])

Initializes a new instance of the [BitArray](#) class that contains bit values copied from the specified array of Booleans.

```
public BitArray (bool[] values);  
new System.Collections.BitArray : bool[] -> System.Collections.BitArray
```

Parameters

values [Boolean\[\]](#)

An array of Booleans to copy.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`values` is `null`.

Remarks

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

## BitArray(Byte[]) BitArray(Byte[])

Initializes a new instance of the [BitArray](#) class that contains bit values copied from the specified array of bytes.

```
public BitArray (byte[] bytes);
new System.Collections.BitArray : byte[] -> System.Collections.BitArray
```

Parameters

bytes [Byte\[\]](#)

An array of bytes containing the values to copy, where each byte represents eight consecutive bits.

Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`bytes` is `null`.

[ArgumentException](#) [ArgumentException](#)

The length of `bytes` is greater than [MaxValue](#).

Remarks

The first byte in the array represents bits 0 through 7, the second byte represents bits 8 through 15, and so on. The Least Significant Bit of each byte represents the lowest index value: "`bytes [0] & 1`" represents bit 0, "`bytes [0] & 2`" represents bit 1, "`bytes [0] & 4`" represents bit 2, and so on.

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

## BitArray(BitArray) BitArray(BitArray)

Initializes a new instance of the [BitArray](#) class that contains bit values copied from the specified [BitArray](#).

```
public BitArray (System.Collections.BitArray bits);
new System.Collections.BitArray : System.Collections.BitArray -> System.Collections.BitArray
```

Parameters

bits [BitArray](#) [BitArray](#)

The [BitArray](#) to copy.

Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`bits` is `null`.

Remarks

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

## BitArray(Int32) BitArray(Int32)

Initializes a new instance of the [BitArray](#) class that can hold the specified number of bit values, which are initially set to `false`.

```
public BitArray (int length);
new System.Collections.BitArray : int -> System.Collections.BitArray
```

#### Parameters

length Int32 Int32

The number of bit values in the new [BitArray](#).

#### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`length` is less than zero.

#### Remarks

This constructor is an  $O(n)$  operation, where `n` is `length`.

## BitArray(Int32[]) BitArray(Int32[])

Initializes a new instance of the [BitArray](#) class that contains bit values copied from the specified array of 32-bit integers.

```
public BitArray (int[] values);
new System.Collections.BitArray : int[] -> System.Collections.BitArray
```

#### Parameters

values Int32[]

An array of integers containing the values to copy, where each integer represents 32 consecutive bits.

#### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`values` is `null`.

[ArgumentException](#) [ArgumentException](#)

The length of `values` is greater than [.MaxValue](#)

#### Remarks

The number in the first `values` array element represents bits 0 through 31, the second number in the array represents bits 32 through 63, and so on. The Least Significant Bit of each integer represents the lowest index value: " `values` [0] & 1" represents bit 0, " `values` [0] & 2" represents bit 1, " `values` [0] & 4" represents bit 2, and so on.

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

## BitArray(Int32, Boolean) BitArray(Int32, Boolean)

Initializes a new instance of the [BitArray](#) class that can hold the specified number of bit values, which are initially set to the specified value.

```
public BitArray (int length, bool defaultValue);
new System.Collections.BitArray : int * bool -> System.Collections.BitArray
```

#### Parameters

`length` Int32 Int32

The number of bit values in the new [BitArray](#).

`defaultValue` Boolean Boolean

The Boolean value to assign to each bit.

**Exceptions**

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`length` is less than zero.

**Remarks**

This constructor is an  $O(n)$  operation, where `n` is `length`.

# BitArray.Clone BitArray.Clone

## In this Article

Creates a shallow copy of the [BitArray](#).

```
public object Clone ();  
  
abstract member Clone : unit -> obj  
override this.Clone : unit -> obj
```

Returns

[Object Object](#)

A shallow copy of the [BitArray](#).

## Remarks

A shallow copy of a collection copies only the elements of the collection, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new collection point to the same objects that the references in the original collection point to.

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](#).

# BitArray.CopyTo BitArray.CopyTo

## In this Article

Copies the entire [BitArray](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

```
public void CopyTo (Array array, int index);  
  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

## Parameters

array [Array](#) [Array](#)

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

index [Int32](#) [Int32](#)

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

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#) [ArgumentException](#)

[array](#) is multidimensional.

-or-

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

[InvalidCastException](#) [InvalidCastException](#)

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

## Examples

The following code example shows how to copy a [BitArray](#) into a one-dimensional [Array](#).

```
using System;  
using System.Collections;  
public class SamplesBitArray {  
  
    public static void Main() {  
  
        // Creates and initializes the source BitArray.  
        BitArray myBA = new BitArray( 4 );  
        myBA[0] = myBA[1] = myBA[2] = myBA[3] = true;  
  
        // Creates and initializes the one-dimensional target Array of type Boolean.  
        bool[] myBoolArray = new bool[8];  
        myBoolArray[0] = false;  
        myBoolArray[1] = false;
```

```

myBoolArray.CopyTo( myBA, 3 );

// Displays the values of the target Array.
Console.WriteLine( "The target Boolean Array contains the following (before and after
copying):" );
PrintValues( myBoolArray );

// Copies the entire source BitArray to the target BitArray, starting at index 3.
myBA.CopyTo( myBoolArray, 3 );

// Displays the values of the target Array.
PrintValues( myBoolArray );

// Creates and initializes the one-dimensional target Array of type integer.
int[] myIntArray = new int[8];
myIntArray[0] = 42;
myIntArray[1] = 43;

// Displays the values of the target Array.
Console.WriteLine( "The target integer Array contains the following (before and after
copying):" );
PrintValues( myIntArray );

// Copies the entire source BitArray to the target BitArray, starting at index 3.
myBA.CopyTo( myIntArray, 3 );

// Displays the values of the target Array.
PrintValues( myIntArray );

// Creates and initializes the one-dimensional target Array of type byte.
Array myByteArray = Array.CreateInstance( typeof(byte), 8 );
myByteArray.SetValue( (byte) 10, 0 );
myByteArray.SetValue( (byte) 11, 1 );

// Displays the values of the target Array.
Console.WriteLine( "The target byte Array contains the following (before and after copying):" );
PrintValues( myByteArray );

// Copies the entire source BitArray to the target BitArray, starting at index 3.
myBA.CopyTo( myByteArray, 3 );

// Displays the values of the target Array.
PrintValues( myByteArray );

// Returns an exception if the array is not of type Boolean, integer or byte.
try {
    Array myStringArray=Array.CreateInstance( typeof(String), 8 );
    myStringArray.SetValue( "Hello", 0 );
    myStringArray.SetValue( "World", 1 );
    myBA.CopyTo( myStringArray, 3 );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

public static void PrintValues( IEnumerable myArr ) {
    foreach ( Object obj in myArr ) {
        Console.Write( "{0,8}", obj );
    }
    Console.WriteLine();
}
}

```

```
/*
This code produces the following output.

The target Boolean Array contains the following (before and after copying):
  False  False  False  False  False  False  False  False
  False  False  False  True   True   True   True   False
The target integer Array contains the following (before and after copying):
  42    43    0     0     0     0     0     0
  42    43    0     15    0     0     0     0
The target byte Array contains the following (before and after copying):
  10    11    0     0     0     0     0     0
  10    11    0     15    0     0     0     0
Exception: System.ArgumentException: Only supported array types for CopyTo on BitArrays are
Boolean[], Int32[] and Byte[].
  at System.Collections.BitArray.CopyTo(Array array, Int32 index)
  at SamplesBitArray.Main()

*/
```

## Remarks

The specified array must be of a compatible type. Only `bool`, `int`, and `byte` types of arrays are supported.

This method uses [Array.Copy](#) to copy the elements.

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

See

[Array](#)[Array](#)

Also

# BitArray.Count BitArray.Count

## In this Article

Gets the number of elements contained in the [BitArray](#).

```
public int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [BitArray](#).

## Remarks

[Length](#) and [Count](#) return the same value. [Length](#) can be set to a specific value, but [Count](#) is read-only.

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

# BitArray.Get BitArray.Get

## In this Article

Gets the value of the bit at a specific position in the [BitArray](#).

```
public bool Get (int index);  
member this.Get : int -> bool
```

### Parameters

index	<a href="#">Int32</a>	<a href="#">Int32</a>
-------	-----------------------	-----------------------

The zero-based index of the value to get.

### Returns

[Boolean](#)

The value of the bit at position `index`.

### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentException](#)

`index` is less than zero.

-or-

`index` is greater than or equal to the number of elements in the [BitArray](#).

## Examples

The following code example shows how to set and get specific elements in a [BitArray](#).

```
using System;  
using System.Collections;  
public class SamplesBitArray {  
  
    public static void Main() {  
  
        // Creates and initializes a BitArray.  
        BitArray myBA = new BitArray( 5 );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "myBA values:" );  
        PrintIndexAndValues( myBA );  
  
        // Sets all the elements to true.  
        myBA.SetAll( true );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "After setting all elements to true," );  
        PrintIndexAndValues( myBA );  
  
        // Sets the last index to false.  
        myBA.Set( myBA.Count - 1, false );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "After setting the last element to false," );  
        PrintIndexAndValues( myBA );  
  
        // Gets the value of the last two elements.  
        Console.WriteLine( "The last two elements are: " );  
        PrintIndexAndValues( myBA );  
    }  
}  
// The output is:  
// myBA values:  
// 0  
// 1  
// 0  
// 1  
// 0  
// After setting all elements to true,  
// 1  
// 1  
// 1  
// 1  
// 1  
// After setting the last element to false,  
// 1  
// 1  
// 1  
// 1  
// 0  
// The last two elements are:  
// 1  
// 0
```

```

        Console.WriteLine( "The last two elements are: " );
        Console.WriteLine( "    at index {0} : {1}", myBA.Count - 2, myBA.Get( myBA.Count - 2 ) );
        Console.WriteLine( "    at index {0} : {1}", myBA.Count - 1, myBA.Get( myBA.Count - 1 ) );
    }

    public static void PrintIndexAndValues( IEnumerable myCol ) {
        int i = 0;
        foreach ( Object obj in myCol ) {
            Console.WriteLine( "    [{0}]:    {1}", i++, obj );
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

myBA values:
[0]: False
[1]: False
[2]: False
[3]: False
[4]: False

After setting all elements to true,
[0]: True
[1]: True
[2]: True
[3]: True
[4]: True

After setting the last element to false,
[0]: True
[1]: True
[2]: True
[3]: True
[4]: False

The last two elements are:
at index 3 : True
at index 4 : False
*/

```

## Remarks

This method is an O(1) operation.

# BitArray.GetEnumerator BitArray.GetEnumerator

## In this Article

Returns an enumerator that iterates through the [BitArray](#).

```
public System.Collections.IEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the entire [BitArray](#).

## 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 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.

This method is an O(1) operation.

See

[IEnumerator](#)[IEnumerator](#)

Also

# BitArray.ICollection.CopyTo

## In this Article

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

## Parameters

array	<a href="#">Array</a>
index	<a href="#">Int32</a>

# BitArray.ICollection.Count

## In this Article

```
int System.Collections.ICollection.Count { get; }
```

## Returns

[Int32](#)

# BitArray.ICollection.IsSynchronized

## In this Article

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

## Returns

[Boolean](#)

# BitArray.ICollection.SyncRoot

## In this Article

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

## Returns

[Object](#)

# BitArray.IsReadOnly BitArray.IsReadOnly

## In this Article

Gets a value indicating whether the [BitArray](#) is read-only.

```
public bool IsReadOnly { get; }  
member this.IsReadOnly : bool
```

Returns

[Boolean](#)

This property is always `false`.

## Remarks

[BitArray](#) implements the [IsReadOnly](#) property because it is required by the [System.Collections.IList](#) 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.

This method is an O(1) operation.

# BitArray.IsSynchronized BitArray.IsSynchronized

## In this Article

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

```
public bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean](#)

This property is always [false](#).

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
BitArray myCollection = new BitArray(64, true);  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

This method is an O(1) operation.

## Remarks

[BitArray](#) implements the [IsSynchronized](#) property because it is required by the [System.Collections.ICollection](#) interface.

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 causes 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.

See

[SyncRoot](#)

Also

# BitArray.Item[Int32] BitArray.Item[Int32]

## In this Article

Gets or sets the value of the bit at a specific position in the [BitArray](#).

```
public bool this[int index] { get; set; }  
member this.Item(int) : bool with get, set
```

### Parameters

index Int32 Int32

The zero-based index of the value to get or set.

### Returns

[Boolean](#) Boolean

The value of the bit at position `index`.

### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` is equal to or greater than `Count`.

## 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 keyword to define the indexers instead of implementing the [Item\[Int32\]](#) property. Visual Basic implements [Item\[Int32\]](#) 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.

See

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

Also

# BitArray.LeftShift BitArray.LeftShift

## In this Article

```
public System.Collections.BitArray LeftShift (int count);  
member this.LeftShift : int -> System.Collections.BitArray
```

## Parameters

count Int32 Int32

## Returns

[BitArray](#) [BitArray](#)

# BitArray.Length BitArray.Length

## In this Article

Gets or sets the number of elements in the [BitArray](#).

```
public int Length { get; set; }  
member this.Length : int with get, set
```

Returns

[Int32](#) [Int32](#)

The number of elements in the [BitArray](#).

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

The property is set to a value that is less than zero.

## Remarks

[Length](#) and [Count](#) return the same value. [Length](#) can be set to a specific value, but [Count](#) is read-only.

If [Length](#) is set to a value that is less than [Count](#), the [BitArray](#) is truncated and the elements after the index `value -1` are deleted.

If [Length](#) is set to a value that is greater than [Count](#), the new elements are set to `false`.

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

# BitArray.Not BitArray.Not

## In this Article

Inverts all the bit values in the current [BitArray](#), so that elements set to `true` are changed to `false`, and elements set to `false` are changed to `true`.

```
public System.Collections.BitArray Not ();
member this.Not : unit -> System.Collections.BitArray
```

## Returns

[BitArray](#) [BitArray](#)

The current instance with inverted bit values.

## Examples

The following code example shows how to apply NOT to a [BitArray](#).

```

using System;
using System.Collections;
public class SamplesBitArray {
    public static void Main() {
        // Creates and initializes two BitArrays of the same size.
        BitArray myBA1 = new BitArray( 4 );
        BitArray myBA2 = new BitArray( 4 );
        myBA1[0] = myBA1[1] = false;
        myBA1[2] = myBA1[3] = true;
        myBA2[0] = myBA2[2] = false;
        myBA2[1] = myBA2[3] = true;

        // Performs a bitwise NOT operation between BitArray instances of the same size.
        Console.WriteLine( "Initial values" );
        Console.Write( "myBA1:" );
        PrintValues( myBA1, 8 );
        Console.Write( "myBA2:" );
        PrintValues( myBA2, 8 );
        Console.WriteLine();

        myBA1.Not();
        myBA2.Not();

        Console.WriteLine( "After NOT" );
        Console.Write( "myBA1:" );
        PrintValues( myBA1, 8 );
        Console.Write( "myBA2:" );
        PrintValues( myBA2, 8 );
        Console.WriteLine();
    }

    public static void PrintValues( IEnumerable myList, int myWidth ) {
        int i = myWidth;
        foreach ( Object obj in myList ) {
            if ( i <= 0 ) {
                i = myWidth;
                Console.WriteLine();
            }
            i--;
            Console.Write( "{0,8}", obj );
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Initial values
myBA1: False False True True
myBA2: False True False True

After NOT
myBA1: True True False False
myBA2: True False True False
*/

```

## Remarks

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

# BitArray.Or BitArray.Or

## In this Article

Performs the bitwise OR operation between the elements of the current [BitArray](#) object and the corresponding elements in the specified array. The current [BitArray](#) object will be modified to store the result of the bitwise OR operation.

```
public System.Collections.BitArray Or (System.Collections.BitArray value);  
member this.Or : System.Collections.BitArray -> System.Collections.BitArray
```

### Parameters

**value** [BitArray](#) [BitArray](#)

The array with which to perform the bitwise OR operation.

### Returns

[BitArray](#) [BitArray](#)

An array containing the result of the bitwise OR operation, which is a reference to the current [BitArray](#) object.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

**value** is **null**.

[ArgumentException](#) [ArgumentException](#)

**value** and the current [BitArray](#) do not have the same number of elements.

## Examples

The following code example shows how to perform the OR operation between two [BitArray](#) objects.

```
using System;  
using System.Collections;  
public class SamplesBitArray {  
  
    public static void Main() {  
  
        // Creates and initializes two BitArrays of the same size.  
        BitArray myBA1 = new BitArray( 4 );  
        BitArray myBA2 = new BitArray( 4 );  
        myBA1[0] = myBA1[1] = false;  
        myBA1[2] = myBA1[3] = true;  
        myBA2[0] = myBA2[2] = false;  
        myBA2[1] = myBA2[3] = true;  
  
        // Performs a bitwise OR operation between BitArray instances of the same size.  
        Console.WriteLine( "Initial values" );  
        Console.Write( "myBA1:" );  
        PrintValues( myBA1, 8 );  
        Console.Write( "myBA2:" );  
        PrintValues( myBA2, 8 );  
        Console.WriteLine();  
  
        Console.WriteLine( "Result" );  
        Console.Write( "OR:" );  
        PrintValues( myBA1.Or( myBA2 ), 8 );  
        Console.WriteLine();
```

```

Console.WriteLine( "After OR" );
Console.Write( "myBA1:" );
PrintValues( myBA1, 8 );
Console.Write( "myBA2:" );
PrintValues( myBA2, 8 );
Console.WriteLine();

// Performing OR between BitArray instances of different sizes returns an exception.
try {
    BitArray myBA3 = new BitArray( 8 );
    myBA3[0] = myBA3[1] = myBA3[2] = myBA3[3] = false;
    myBA3[4] = myBA3[5] = myBA3[6] = myBA3[7] = true;
    myBA1.Or( myBA3 );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

public static void PrintValues( IEnumerable myList, int myWidth ) {
    int i = myWidth;
    foreach ( Object obj in myList ) {
        if ( i <= 0 ) {
            i = myWidth;
            Console.WriteLine();
        }
        i--;
        Console.Write( "{0,8}", obj );
    }
    Console.WriteLine();
}

/*
This code produces the following output.

Initial values
myBA1:  False  False  True  True
myBA2:  False  True  False  True

Result
OR:  False  True  True  True

After OR
myBA1:  False  True  True  True
myBA2:  False  True  False  True

Exception: System.ArgumentException: Array lengths must be the same.
  at System.Collections.BitArray.Or(BitArray value)
  at SamplesBitArray.Main()
*/

```

## Remarks

The bitwise OR operation returns `true` if one or both operands are `true`, and returns `false` if both operands are `false`.

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

# BitArray.RightShift BitArray.RightShift

## In this Article

```
public System.Collections.BitArray RightShift (int count);  
member this.RightShift : int -> System.Collections.BitArray
```

## Parameters

count Int32 Int32

## Returns

[BitArray](#) [BitArray](#)

# BitArray.Set BitArray.Set

## In this Article

Sets the bit at a specific position in the [BitArray](#) to the specified value.

```
public void Set (int index, bool value);  
member this.Set : int * bool -> unit
```

## Parameters

index Int32 Int32

The zero-based index of the bit to set.

value Boolean Boolean

The Boolean value to assign to the bit.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` is greater than or equal to the number of elements in the [BitArray](#).

## Examples

The following code example shows how to set and get specific elements in a [BitArray](#).

```
using System;  
using System.Collections;  
public class SamplesBitArray {  
  
    public static void Main() {  
  
        // Creates and initializes a BitArray.  
        BitArray myBA = new BitArray( 5 );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "myBA values:" );  
        PrintIndexAndValues( myBA );  
  
        // Sets all the elements to true.  
        myBA.SetAll( true );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "After setting all elements to true," );  
        PrintIndexAndValues( myBA );  
  
        // Sets the last index to false.  
        myBA.Set( myBA.Count - 1, false );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "After setting the last element to false," );  
        PrintIndexAndValues( myBA );  
  
        // Gets the value of the last two elements.  
        Console.WriteLine( "The last two elements are: " );  
        Console.WriteLine( "    at index {0} : {1} ". mvBA.Count - 2, mvBA.Get( mvBA.Count - 2 ) );  
    }  
}
```

```

        Console.WriteLine( "      at index {0} : {1}", myBA.Count - 1, myBA.Get( myBA.Count - 1 ) );
    }

public static void PrintIndexAndValues( IEnumerable myCol ) {
    int i = 0;
    foreach ( Object obj in myCol ) {
        Console.WriteLine( "      [{0}]:     {1}", i++, obj );
    }
    Console.WriteLine();
}

/*
This code produces the following output.

myBA values:
[0]: False
[1]: False
[2]: False
[3]: False
[4]: False

After setting all elements to true,
[0]: True
[1]: True
[2]: True
[3]: True
[4]: True

After setting the last element to false,
[0]: True
[1]: True
[2]: True
[3]: True
[4]: False

The last two elements are:
at index 3 : True
at index 4 : False
*/

```

## Remarks

This method is an O(1) operation.

# BitArray.SetAll BitArray.SetAll

## In this Article

Sets all bits in the [BitArray](#) to the specified value.

```
public void SetAll (bool value);  
member this.SetAll : bool -> unit
```

## Parameters

value	Boolean Boolean
-------	-----------------

The Boolean value to assign to all bits.

## Examples

The following code example shows how to set and get specific elements in a [BitArray](#).

```
using System;  
using System.Collections;  
public class SamplesBitArray {  
  
    public static void Main() {  
  
        // Creates and initializes a BitArray.  
        BitArray myBA = new BitArray( 5 );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "myBA values:" );  
        PrintIndexAndValues( myBA );  
  
        // Sets all the elements to true.  
        myBA.SetAll( true );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "After setting all elements to true," );  
        PrintIndexAndValues( myBA );  
  
        // Sets the last index to false.  
        myBA.Set( myBA.Count - 1, false );  
  
        // Displays the properties and values of the BitArray.  
        Console.WriteLine( "After setting the last element to false," );  
        PrintIndexAndValues( myBA );  
  
        // Gets the value of the last two elements.  
        Console.WriteLine( "The last two elements are: " );  
        Console.WriteLine( "    at index {0} : {1}", myBA.Count - 2, myBA.Get( myBA.Count - 2 ) );  
        Console.WriteLine( "    at index {0} : {1}", myBA.Count - 1, myBA.Get( myBA.Count - 1 ) );  
    }  
  
    public static void PrintIndexAndValues( IEnumerable myCol ) {  
        int i = 0;  
        foreach ( Object obj in myCol ) {  
            Console.WriteLine( "    [{0}]: {1}", i++, obj );  
        }  
        Console.WriteLine();  
    }  
}
```

```
/*
This code produces the following output.
```

```
myBA values:  
[0]: False  
[1]: False  
[2]: False  
[3]: False  
[4]: False
```

```
After setting all elements to true,
```

```
[0]: True  
[1]: True  
[2]: True  
[3]: True  
[4]: True
```

```
After setting the last element to false,
```

```
[0]: True  
[1]: True  
[2]: True  
[3]: True  
[4]: False
```

```
The last two elements are:
```

```
at index 3 : True  
at index 4 : False
```

```
*/
```

## Remarks

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

# BitArray.SyncRoot BitArray.SyncRoot

## In this Article

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

```
public object SyncRoot { get; }  
member this.SyncRoot : obj
```

Returns

[Object Object](#)

An object that can be used to synchronize access to the [BitArray](#).

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
BitArray myCollection = new BitArray(64, true);  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

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

## Remarks

Derived classes can provide their own synchronized version of the [BitArray](#) using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) of the [BitArray](#), not directly on the [BitArray](#). This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [BitArray](#) object.

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 causes 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.

See

[IsSynchronized](#)[IsSynchronized](#)

Also

# BitArray.Xor BitArray.Xor

## In this Article

Performs the bitwise exclusive OR operation between the elements of the current [BitArray](#) object against the corresponding elements in the specified array. The current [BitArray](#) object will be modified to store the result of the bitwise exclusive OR operation.

```
public System.Collections.BitArray Xor (System.Collections.BitArray value);  
member this.Xor : System.Collections.BitArray -> System.Collections.BitArray
```

### Parameters

value [BitArray](#) [BitArray](#)

The array with which to perform the bitwise exclusive OR operation.

### Returns

[BitArray](#) [BitArray](#)

An array containing the result of the bitwise exclusive OR operation, which is a reference to the current [BitArray](#) object.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`value` is `null`.

[ArgumentException](#) [ArgumentException](#)

`value` and the current [BitArray](#) do not have the same number of elements.

## Examples

The following code example shows how to perform the XOR operation between two [BitArray](#) objects.

```
using System;  
using System.Collections;  
public class SamplesBitArray {  
  
    public static void Main() {  
  
        // Creates and initializes two BitArrays of the same size.  
        BitArray myBA1 = new BitArray( 4 );  
        BitArray myBA2 = new BitArray( 4 );  
        myBA1[0] = myBA1[1] = false;  
        myBA1[2] = myBA1[3] = true;  
        myBA2[0] = myBA2[2] = false;  
        myBA2[1] = myBA2[3] = true;  
  
        // Performs a bitwise XOR operation between BitArray instances of the same size.  
        Console.WriteLine( "Initial values" );  
        Console.Write( "myBA1:" );  
        PrintValues( myBA1, 8 );  
        Console.Write( "myBA2:" );  
        PrintValues( myBA2, 8 );  
        Console.WriteLine();  
  
        Console.WriteLine( "Result" );  
        Console.Write( "XOR:" );  
        PrintValues( myBA1.Xor( myBA2 ), 8 );  
        Console.WriteLine();
```

```

Console.WriteLine( "After XOR" );
Console.Write( "myBA1:" );
PrintValues( myBA1, 8 );
Console.Write( "myBA2:" );
PrintValues( myBA2, 8 );
Console.WriteLine();

// Performing XOR between BitArray instances of different sizes returns an exception.
try {
    BitArray myBA3 = new BitArray( 8 );
    myBA3[0] = myBA3[1] = myBA3[2] = myBA3[3] = false;
    myBA3[4] = myBA3[5] = myBA3[6] = myBA3[7] = true;
    myBA1.Xor( myBA3 );
} catch ( Exception myException ) {
    Console.WriteLine("Exception: " + myException.ToString());
}
}

public static void PrintValues( IEnumerable myList, int myWidth ) {
    int i = myWidth;
    foreach ( Object obj in myList ) {
        if ( i <= 0 ) {
            i = myWidth;
            Console.WriteLine();
        }
        i--;
        Console.Write( "{0,8}", obj );
    }
    Console.WriteLine();
}

}

/*
This code produces the following output.

Initial values
myBA1:  False  False  True  True
myBA2:  False  True  False  True

Result
XOR:  False  True  True  False

After XOR
myBA1:  False  True  True  False
myBA2:  False  True  False  True

Exception: System.ArgumentException: Array lengths must be the same.
at System.Collections.BitArray.Xor(BitArray value)
at SamplesBitArray.Main()

*/

```

## Remarks

The bitwise exclusive OR operation returns `true` if exactly one operand is `true`, and returns `false` if both operands have the same Boolean value.

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

# CaseInsensitiveComparer CaseInsensitiveComparer Class

Compares two objects for equivalence, ignoring the case of strings.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public class CaseInsensitiveComparer : System.Collections.IComparer

type CaseInsensitiveComparer = class
    interface IComparer
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

[CaseInsensitiveComparer](#) implements the [IComparer](#) interface supporting case-insensitive comparisons on strings, just as [CaseInsensitiveHashCodeProvider](#) implements the [IHashCodeProvider](#) interface supporting case-insensitive comparisons on strings.

### Important

We don't recommend that you use the [CaseInsensitiveComparer](#) class for new development. Instead, we recommend that you use the [System.StringComparer](#) object returned by the [StringComparer.CurrentCultureIgnoreCase](#), [StringComparer.InvariantCultureIgnoreCase](#), or [StringComparer.OrdinalIgnoreCase](#) property.

The [Comparer](#) class is the default implementation of the [IComparer](#) interface and performs case-sensitive string comparisons.

The objects used as keys by a [Hashtable](#) are required to override the [Object.GetHashCode](#) method (or the [IHashCodeProvider](#) interface) and the [Object.Equals](#) method (or the [IComparer](#) interface). The implementation of both methods or interfaces must handle case sensitivity the same way; otherwise, the [Hashtable](#) might behave incorrectly. For example, when creating a [Hashtable](#), you must use this class with the [CaseInsensitiveHashCodeProvider](#) class or any case-insensitive [IHashCodeProvider](#) implementation.

String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

## Constructors

[CaseInsensitiveComparer\(\)](#)

[CaseInsensitiveComparer\(\)](#)

Initializes a new instance of the [CaseInsensitiveComparer](#) class using the [CurrentCulture](#) of the current thread.

[CaseInsensitiveComparer\(CultureInfo\)](#)

[CaseInsensitiveComparer\(CultureInfo\)](#)

Initializes a new instance of the [CaseInsensitiveComparer](#) class using the specified [CultureInfo](#).

## Properties

**Default**

**Default**

Gets an instance of [CaseInsensitiveComparer](#) that is associated with the [CurrentCulture](#) of the current thread and that is always available.

**DefaultInvariant**

**DefaultInvariant**

Gets an instance of [CaseInsensitiveComparer](#) that is associated with [InvariantCulture](#) and that is always available.

## Methods

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

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

Performs a case-insensitive comparison of two objects of the same type and returns a value indicating whether one is less than, equal to, or greater than the other.

## See Also

[IComparer](#) [IComparer](#)

[CultureInfo](#) [CultureInfo](#)

[IComparer](#) [IComparer](#)

# CaseInsensitiveComparer CaseInsensitiveComparer

In this Article

## Overloads

<code>CaseInsensitiveComparer()</code>	Initializes a new instance of the <code>CaseInsensitiveComparer</code> class using the <code>CurrentCulture</code> of the current thread.
<code>CaseInsensitiveComparer(CultureInfo) CaseInsensitiveComparer(CultureInfo)</code>	Initializes a new instance of the <code>CaseInsensitiveComparer</code> class using the specified <code>CultureInfo</code> .

## CaseInsensitiveComparer()

Initializes a new instance of the `CaseInsensitiveComparer` class using the `CurrentCulture` of the current thread.

```
public CaseInsensitiveComparer ();
```

## Examples

The following code example creates a case-sensitive hash table and a case-insensitive hash table and demonstrates the difference in their behavior, even if both contain the same elements.

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesHashtable {
    public static void Main() {
        // Create a Hashtable using the default hash code provider and the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the culture of the current thread.
        Hashtable myHT2 = new Hashtable( new CaseInsensitiveHashCodeProvider(), new
        CaseInsensitiveComparer() );
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the InvariantCulture.
        Hashtable myHT3 = new Hashtable( CaseInsensitiveHashCodeProvider.DefaultInvariant,
        CaseInsensitiveComparer.DefaultInvariant );
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the Turkish culture (tr-TR), where "I" is not the uppercase version of "i".
        CultureInfo myCul = new CultureInfo( "tr-TR" );
        Hashtable myHT4 = new Hashtable( new CaseInsensitiveHashCodeProvider( myCul ), new
        CaseInsensitiveComparer( myCul ) );
        myHT4.Add("FIRST", "Hello");
        myHT4.Add("SECOND", "World");
        myHT4.Add("THIRD", "!");

        // Search for a key in each hashtable.
        Console.WriteLine( "first is in myHT1: {0}", myHT1.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT2: {0}", myHT2.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT3: {0}", myHT3.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT4: {0}", myHT4.ContainsKey( "first" ) );
    }
}

/*
This code produces the following output. Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: True
first is in myHT4: False
*/

```

## Remarks

When the [CaseInsensitiveComparer](#) instance is created using this constructor, the [Thread.CurrentCulture](#) of the current

thread is saved. Comparison procedures use the saved culture to determine the sort order and casing rules; therefore, string comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

See

[CultureInfo](#)

Also

[Performing Culture-Insensitive String Operations in Collections](#)

## **CaseInsensitiveComparer(CultureInfo)**

## **CaseInsensitiveComparer(CultureInfo)**

Initializes a new instance of the [CaseInsensitiveComparer](#) class using the specified [CultureInfo](#).

```
public CaseInsensitiveComparer (System.Globalization.CultureInfo culture);  
new System.Collections.CaseInsensitiveComparer : System.Globalization.CultureInfo ->  
System.Collections.CaseInsensitiveComparer
```

Parameters

culture

[CultureInfo](#)

The [CultureInfo](#) to use for the new [CaseInsensitiveComparer](#).

Exceptions

[ArgumentNullException](#)

`culture` is `null`.

Examples

The following code example creates a case-sensitive hash table and a case-insensitive hash table and demonstrates the difference in their behavior, even if both contain the same elements.

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesHashtable {
    public static void Main() {
        // Create a Hashtable using the default hash code provider and the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the culture of the current thread.
        Hashtable myHT2 = new Hashtable( new CaseInsensitiveHashCodeProvider(), new
        CaseInsensitiveComparer() );
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the InvariantCulture.
        Hashtable myHT3 = new Hashtable( CaseInsensitiveHashCodeProvider.DefaultInvariant,
        CaseInsensitiveComparer.DefaultInvariant );
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the Turkish culture (tr-TR), where "I" is not the uppercase version of "i".
        CultureInfo myCul = new CultureInfo( "tr-TR" );
        Hashtable myHT4 = new Hashtable( new CaseInsensitiveHashCodeProvider( myCul ), new
        CaseInsensitiveComparer( myCul ) );
        myHT4.Add("FIRST", "Hello");
        myHT4.Add("SECOND", "World");
        myHT4.Add("THIRD", "!");

        // Search for a key in each hashtable.
        Console.WriteLine( "first is in myHT1: {0}", myHT1.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT2: {0}", myHT2.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT3: {0}", myHT3.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT4: {0}", myHT4.ContainsKey( "first" ) );
    }
}

/*
This code produces the following output. Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: True
first is in myHT4: False
*/

```

## Remarks

Comparison procedures use the specified [System.Globalization.CultureInfo](#) to determine the sort order and casing

rules. String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

See

[CultureInfo](#)

Also

[Performing Culture-Insensitive String Operations in Collections](#)

# CASEINSENSITIVECOMPARER.COMPARER.CASEINSENSITIVECOMPARER.COMPARER.COMPARER

## In this Article

Performs a case-insensitive comparison of two objects of the same type and returns a value indicating whether one is less than, equal to, or greater than the other.

```
public int Compare (object a, object b);  
abstract member Compare : obj * obj -> int  
override this.Compare : obj * obj -> int
```

## Parameters

a	Object Object
The first object to compare.	
b	Object Object
The second object to compare.	
Returns	

## Int32 Int32

A signed integer that indicates the relative values of `a` and `b`, as shown in the following table.

Value	Meaning
Less than zero	<code>a</code> is less than <code>b</code> , with casing ignored.
Zero	<code>a</code> equals <code>b</code> , with casing ignored.
Greater than zero	<code>a</code> is greater than <code>b</code> , with casing ignored.

## Exceptions

### ArgumentException ArgumentException

Neither `a` nor `b` implements the `IComparable` interface.

-or-

`a` and `b` are of different types.

## Remarks

If `a` and `b` are both strings, this method uses `CompareInfo.Compare` to compare the strings with the casing ignored; otherwise, it uses the `IComparable` implementation of either object. That is, if `a` implements `IComparable`, then this method returns the result of `a.CompareTo(b)`; otherwise, if `b` implements `IComparable`, then it returns the negated result of `b.CompareTo(a)`.

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

String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the `System.Globalization` namespace and [Globalization and Localization](#).

See

Also

[CultureInfo](#)[CultureInfo](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# CASEINSENSITIVECOMPARER.DEFAULT CASEINSENSITIVE COMPARER.DEFAULT

## In this Article

Gets an instance of [CaseInsensitiveComparer](#) that is associated with the [CurrentCulture](#) of the current thread and that is always available.

```
public static System.Collections.CaseInsensitiveComparer Default { get; }  
member this.Default : System.Collections.CaseInsensitiveComparer
```

## Returns

[CaseInsensitiveComparer](#) [CaseInsensitiveComparer](#)

An instance of [CaseInsensitiveComparer](#) that is associated with the [CurrentCulture](#) of the current thread.

## Remarks

When the [CaseInsensitiveComparer](#) instance is created using the parameterless constructor, the [Thread.CurrentCulture](#) of the current thread is saved. Comparison procedures use the saved culture to determine the sort order and casing rules; therefore, string comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

### See

[Performing Culture-Insensitive String Operations in Collections](#)

### Also

# CaseInsensitiveComparer.DefaultInvariant CaseInsensitiveComparer.DefaultInvariant

## In this Article

Gets an instance of [CaseInsensitiveComparer](#) that is associated with [InvariantCulture](#) and that is always available.

```
public static System.Collections.CaseInsensitiveComparer DefaultInvariant { get; }  
member this.DefaultInvariant : System.Collections.CaseInsensitiveComparer
```

Returns

[CaseInsensitiveComparer](#) [CaseInsensitiveComparer](#)

An instance of [CaseInsensitiveComparer](#) that is associated with [InvariantCulture](#).

## Examples

The following code example creates a case-sensitive hash table and a case-insensitive hash table and demonstrates the difference in their behavior, even if both contain the same elements.

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesHashtable {
    public static void Main() {
        // Create a Hashtable using the default hash code provider and the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the culture of the current thread.
        Hashtable myHT2 = new Hashtable( new CaseInsensitiveHashCodeProvider(), new
        CaseInsensitiveComparer() );
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the InvariantCulture.
        Hashtable myHT3 = new Hashtable( CaseInsensitiveHashCodeProvider.DefaultInvariant,
        CaseInsensitiveComparer.DefaultInvariant );
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the Turkish culture (tr-TR), where "I" is not the uppercase version of "i".
        CultureInfo myCul = new CultureInfo( "tr-TR" );
        Hashtable myHT4 = new Hashtable( new CaseInsensitiveHashCodeProvider( myCul ), new
        CaseInsensitiveComparer( myCul ) );
        myHT4.Add("FIRST", "Hello");
        myHT4.Add("SECOND", "World");
        myHT4.Add("THIRD", "!");

        // Search for a key in each hashtable.
        Console.WriteLine( "first is in myHT1: {0}", myHT1.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT2: {0}", myHT2.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT3: {0}", myHT3.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT4: {0}", myHT4.ContainsKey( "first" ) );
    }
}

/*
This code produces the following output. Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: True
first is in myHT4: False
*/

```

## Remarks

Comparison procedures use the [CultureInfo.InvariantCulture](#) to determine the sort order and casing rules. String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

See

[Performing Culture-Insensitive String Operations in Collections](#)

Also

# CaseInsensitiveHashCodeProvider Class

Supplies a hash code for an object, using a hashing algorithm that ignores the case of strings.

## Declaration

```
[System.Obsolete("Please use StringComparer instead.")]
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public class CaseInsensitiveHashCodeProvider : System.Collections.IHashCodeProvider

type CaseInsensitiveHashCodeProvider = class
    interface IHashCodeProvider
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

[CaseInsensitiveHashCodeProvider](#) implements the [IHashCodeProvider](#) interface supporting case-insensitive comparisons on strings, just as [CaseInsensitiveComparer](#) implements the [IComparer](#) interface supporting case-insensitive comparisons on strings.

### Important

We don't recommend that you use the [CaseInsensitiveHashCodeProvider](#) class for new development. Instead, we recommend that you use the [System.StringComparer](#) object returned by the [StringComparer.CurrentCultureIgnoreCase](#), [StringComparer.InvariantCultureIgnoreCase](#), or [StringComparer.OrdinalIgnoreCase](#) property.

The objects used as keys by a [Hashtable](#) are required to override the [Object.GetHashCode](#) method (or the [IHashCodeProvider](#) interface) and the [Object.Equals](#) method (or the [IComparer](#) interface). The implementation of both methods or interfaces must handle case sensitivity the same way; otherwise, the [Hashtable](#) might behave incorrectly. For example, when creating a [Hashtable](#), you must use this class with the [CaseInsensitiveComparer](#) class or any case-insensitive [IComparer](#) implementation.

## Constructors

[CaseInsensitiveHashCodeProvider\(\)](#)

[CaseInsensitiveHashCodeProvider\(\)](#)

Initializes a new instance of the [CaseInsensitiveHashCodeProvider](#) class using the [CurrentCulture](#) of the current thread.

[CaseInsensitiveHashCodeProvider\(CultureInfo\)](#)

[CaseInsensitiveHashCodeProvider\(CultureInfo\)](#)

Initializes a new instance of the [CaseInsensitiveHashCodeProvider](#) class using the specified [CultureInfo](#).

## Properties

**Default**

**Default**

Gets an instance of [CaseInsensitiveHashCodeProvider](#) that is associated with the [CurrentCulture](#) of the current thread and that is always available.

**DefaultInvariant**

**DefaultInvariant**

Gets an instance of [CaseInsensitiveHashCodeProvider](#) that is associated with [InvariantCulture](#) and that is always available.

## Methods

[GetHashCode\(Object\)](#)

[GetHashCode\(Object\)](#)

Returns a hash code for the given object, using a hashing algorithm that ignores the case of strings.

## See Also

[IHashCodeProvider](#) [IHashCodeProvider](#)

[CultureInfo](#) [CultureInfo](#)

[IHashCodeProvider](#) [IHashCodeProvider](#)

# CaseInsensitiveHashCodeProvider CaseInsensitiveHashCodeProvider

In this Article

## Overloads

<a href="#">CaseInsensitiveHashCodeProvider()</a>	Initializes a new instance of the <a href="#">CaseInsensitiveHashCodeProvider</a> class using the <a href="#">CurrentCulture</a> of the current thread.
<a href="#">CaseInsensitiveHashCodeProvider(CultureInfo)</a> <a href="#">CaseInsensitiveHashCodeProvider(CultureInfo)</a>	Initializes a new instance of the <a href="#">CaseInsensitiveHashCodeProvider</a> class using the specified <a href="#">CultureInfo</a> .

### CaseInsensitiveHashCodeProvider()

Initializes a new instance of the [CaseInsensitiveHashCodeProvider](#) class using the [CurrentCulture](#) of the current thread.

```
public CaseInsensitiveHashCodeProvider ();
```

#### Examples

The following code example creates a case-sensitive hash table and a case-insensitive hash table and demonstrates the difference in their behavior, even if both contain the same elements.

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesHashtable {
    public static void Main() {
        // Create a Hashtable using the default hash code provider and the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the culture of the current thread.
        Hashtable myHT2 = new Hashtable( new CaseInsensitiveHashCodeProvider(), new
        CaseInsensitiveComparer() );
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the InvariantCulture.
        Hashtable myHT3 = new Hashtable( CaseInsensitiveHashCodeProvider.DefaultInvariant,
        CaseInsensitiveComparer.DefaultInvariant );
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the Turkish culture (tr-TR), where "I" is not the uppercase version of "i".
        CultureInfo myCul = new CultureInfo( "tr-TR" );
        Hashtable myHT4 = new Hashtable( new CaseInsensitiveHashCodeProvider( myCul ), new
        CaseInsensitiveComparer( myCul ) );
        myHT4.Add("FIRST", "Hello");
        myHT4.Add("SECOND", "World");
        myHT4.Add("THIRD", "!");

        // Search for a key in each hashtable.
        Console.WriteLine( "first is in myHT1: {0}", myHT1.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT2: {0}", myHT2.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT3: {0}", myHT3.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT4: {0}", myHT4.ContainsKey( "first" ) );
    }
}

/*
This code produces the following output. Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: True
first is in myHT4: False
*/

```

## Remarks

When the [CaseInsensitiveHashCodeProvider](#) instance is created using this constructor, the [Thread.CurrentCulture](#) of

the current thread is saved. Comparison procedures use the saved culture to determine the casing rules; therefore, hash code comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

See

[CultureInfo](#)

Also

[Performing Culture-Insensitive String Operations in Collections](#)

## **CaseInsensitiveHashCodeProvider(CultureInfo)**

## **CaseInsensitiveHashCodeProvider(CultureInfo)**

Initializes a new instance of the [CaseInsensitiveHashCodeProvider](#) class using the specified [CultureInfo](#).

```
public CaseInsensitiveHashCodeProvider (System.Globalization.CultureInfo culture);  
new System.Collections.CaseInsensitiveHashCodeProvider : System.Globalization.CultureInfo ->  
System.Collections.CaseInsensitiveHashCodeProvider
```

Parameters

culture

[CultureInfo](#) [CultureInfo](#)

The [CultureInfo](#) to use for the new [CaseInsensitiveHashCodeProvider](#).

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`culture` is `null`.

Examples

The following code example creates a case-sensitive hash table and a case-insensitive hash table and demonstrates the difference in their behavior, even if both contain the same elements.

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesHashtable {
    public static void Main() {
        // Create a Hashtable using the default hash code provider and the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the culture of the current thread.
        Hashtable myHT2 = new Hashtable( new CaseInsensitiveHashCodeProvider(), new
        CaseInsensitiveComparer() );
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the InvariantCulture.
        Hashtable myHT3 = new Hashtable( CaseInsensitiveHashCodeProvider.DefaultInvariant,
        CaseInsensitiveComparer.DefaultInvariant );
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the Turkish culture (tr-TR), where "I" is not the uppercase version of "i".
        CultureInfo myCul = new CultureInfo( "tr-TR" );
        Hashtable myHT4 = new Hashtable( new CaseInsensitiveHashCodeProvider( myCul ), new
        CaseInsensitiveComparer( myCul ) );
        myHT4.Add("FIRST", "Hello");
        myHT4.Add("SECOND", "World");
        myHT4.Add("THIRD", "!");

        // Search for a key in each hashtable.
        Console.WriteLine( "first is in myHT1: {0}", myHT1.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT2: {0}", myHT2.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT3: {0}", myHT3.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT4: {0}", myHT4.ContainsKey( "first" ) );
    }
}

/*
This code produces the following output. Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: True
first is in myHT4: False
*/

```

## Remarks

Comparison procedures use the specified [System.Globalization.CultureInfo](#) to determine the casing rules. Hash code

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

See

[CultureInfo](#)

Also

[Performing Culture-Insensitive String Operations in Collections](#)

# CaseInsensitiveHashCodeProvider.Default CaseInsensitiveHashCodeProvider.Default

## In this Article

Gets an instance of [CaseInsensitiveHashCodeProvider](#) that is associated with the [CurrentCulture](#) of the current thread and that is always available.

```
public static System.Collections.CaseInsensitiveHashCodeProvider Default { get; }  
member this.Default : System.Collections.CaseInsensitiveHashCodeProvider
```

## Returns

[CaseInsensitiveHashCodeProvider](#) [CaseInsensitiveHashCodeProvider](#)

An instance of [CaseInsensitiveHashCodeProvider](#) that is associated with the [CurrentCulture](#) of the current thread.

## Remarks

When the [CaseInsensitiveHashCodeProvider](#) instance is created using the parameterless constructor, the [Thread.CurrentCulture](#) of the current thread is saved. Comparison procedures use the saved culture to determine the casing rules; therefore, hash code comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

### See

[Performing Culture-Insensitive String Operations in Collections](#)

### Also

# CaseInsensitiveHashCodeProvider.DefaultInvariant Case InsensitiveHashCodeProvider.DefaultInvariant

## In this Article

Gets an instance of [CaseInsensitiveHashCodeProvider](#) that is associated with [InvariantCulture](#) and that is always available.

```
public static System.Collections.CaseInsensitiveHashCodeProvider DefaultInvariant { get; }  
member this.DefaultInvariant : System.Collections.CaseInsensitiveHashCodeProvider
```

## Returns

[CaseInsensitiveHashCodeProvider](#) [CaseInsensitiveHashCodeProvider](#)

An instance of [CaseInsensitiveHashCodeProvider](#) that is associated with [InvariantCulture](#).

## Examples

The following code example creates a case-sensitive hash table and a case-insensitive hash table and demonstrates the difference in their behavior, even if both contain the same elements.

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesHashtable {
    public static void Main() {
        // Create a Hashtable using the default hash code provider and the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the culture of the current thread.
        Hashtable myHT2 = new Hashtable( new CaseInsensitiveHashCodeProvider(), new
        CaseInsensitiveComparer() );
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the InvariantCulture.
        Hashtable myHT3 = new Hashtable( CaseInsensitiveHashCodeProvider.DefaultInvariant,
        CaseInsensitiveComparer.DefaultInvariant );
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Create a Hashtable using a case-insensitive code provider and a case-insensitive comparer,
        // based on the Turkish culture (tr-TR), where "I" is not the uppercase version of "i".
        CultureInfo myCul = new CultureInfo( "tr-TR" );
        Hashtable myHT4 = new Hashtable( new CaseInsensitiveHashCodeProvider( myCul ), new
        CaseInsensitiveComparer( myCul ) );
        myHT4.Add("FIRST", "Hello");
        myHT4.Add("SECOND", "World");
        myHT4.Add("THIRD", "!");

        // Search for a key in each hashtable.
        Console.WriteLine( "first is in myHT1: {0}", myHT1.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT2: {0}", myHT2.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT3: {0}", myHT3.ContainsKey( "first" ) );
        Console.WriteLine( "first is in myHT4: {0}", myHT4.ContainsKey( "first" ) );
    }
}

/*
This code produces the following output. Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: True
first is in myHT4: False
*/

```

## Remarks

Comparison procedures use the [CultureInfo.InvariantCulture](#) to determine the casing rules. Hash code comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

See

[Performing Culture-Insensitive String Operations in Collections](#)

Also

# CaseInsensitiveHashCodeProvider.GetHashCode Case InsensitiveHashCodeProvider.GetHashCode

## In this Article

Returns a hash code for the given object, using a hashing algorithm that ignores the case of strings.

```
public int GetHashCode (object obj);  
override this.GetHashCode : obj -> int
```

### Parameters

obj Object Object

The [Object](#) for which a hash code is to be returned.

### Returns

[Int32 Int32](#)

A hash code for the given object, using a hashing algorithm that ignores the case of strings.

### Exceptions

[ArgumentNullException ArgumentNullException](#)

obj is `null`.

## Remarks

The return value from this method must not be persisted for two reasons. First, the hash function of a class might be altered to generate a better distribution, thereby rendering any values from the old hash function useless. Second, the default implementation of this class does not guarantee that the same value will be returned by different instances.

### See

[GetHashCode\(\) GetHashCode\(\)](#)

### Also

[Performing Culture-Insensitive String Operations in Collections](#)

# CollectionBase CollectionBase Class

Provides the `abstract` base class for a strongly typed collection.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public abstract class CollectionBase : System.Collections.IList

type CollectionBase = class
    interface IList
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

`Object` `Object`

## Remarks

**Important**

We don't recommend that you use the `CollectionBase` class for new development. Instead, we recommend that you use the generic `Collection<T>` class. For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

A `CollectionBase` instance is always modifiable. See `ReadOnlyCollectionBase` for a read-only version of this class.

The capacity of a `CollectionBase` is the number of elements the `CollectionBase` can hold. As elements are added to a `CollectionBase`, the capacity is automatically increased as required through reallocation. The capacity can be decreased by setting the `Capacity` property explicitly.

## Constructors

`CollectionBase()`

`CollectionBase()`

Initializes a new instance of the `CollectionBase` class with the default initial capacity.

`CollectionBase(Int32)`

`CollectionBase(Int32)`

Initializes a new instance of the `CollectionBase` class with the specified capacity.

## Properties

`Capacity`

`Capacity`

Gets or sets the number of elements that the `CollectionBase` can contain.

`Count`

`Count`

Gets the number of elements contained in the [CollectionBase](#) instance. This property cannot be overridden.

`InnerList`

`InnerList`

Gets an [ArrayList](#) containing the list of elements in the [CollectionBase](#) instance.

`List`

`List`

Gets an [IList](#) containing the list of elements in the [CollectionBase](#) instance.

## Methods

`Clear()`

`Clear()`

Removes all objects from the [CollectionBase](#) instance. This method cannot be overridden.

`GetEnumerator()`

`GetEnumerator()`

Returns an enumerator that iterates through the [CollectionBase](#) instance.

`OnClear()`

`OnClear()`

Performs additional custom processes when clearing the contents of the [CollectionBase](#) instance.

`OnClearComplete()`

`OnClearComplete()`

Performs additional custom processes after clearing the contents of the [CollectionBase](#) instance.

`OnInsert(Int32, Object)`

`OnInsert(Int32, Object)`

Performs additional custom processes before inserting a new element into the [CollectionBase](#) instance.

`OnInsertComplete(Int32, Object)`

`OnInsertComplete(Int32, Object)`

Performs additional custom processes after inserting a new element into the [CollectionBase](#) instance.

```
OnRemove(Int32, Object)  
OnRemove(Int32, Object)
```

Performs additional custom processes when removing an element from the [CollectionBase](#) instance.

```
OnRemoveComplete(Int32, Object)  
OnRemoveComplete(Int32, Object)
```

Performs additional custom processes after removing an element from the [CollectionBase](#) instance.

```
OnSet(Int32, Object, Object)  
OnSet(Int32, Object, Object)
```

Performs additional custom processes before setting a value in the [CollectionBase](#) instance.

```
OnSetComplete(Int32, Object, Object)  
OnSetComplete(Int32, Object, Object)
```

Performs additional custom processes after setting a value in the [CollectionBase](#) instance.

```
OnValidate(Object)  
OnValidate(Object)
```

Performs additional custom processes when validating a value.

```
RemoveAt(Int32)  
RemoveAt(Int32)
```

Removes the element at the specified index of the [CollectionBase](#) instance. This method is not overridable.

```
ICollection.CopyTo(Array, Int32)  
ICollection.CopyTo(Array, Int32)
```

Copies the entire [CollectionBase](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

```
ICollection.IsSynchronized  
ICollection.IsSynchronized
```

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

```
ICollection.SyncRoot  
ICollection.SyncRoot
```

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

```
IList.Add(Object)  
IList.Add(Object)
```

Adds an object to the end of the [CollectionBase](#).

```
IList.Contains(Object)  
IList.Contains(Object)
```

Determines whether the [CollectionBase](#) contains a specific element.

```
IList.IndexOf(Object)  
IList.IndexOf(Object)
```

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the entire [CollectionBase](#).

```
IList.Insert(Int32, Object)  
IList.Insert(Int32, Object)
```

Inserts an element into the [CollectionBase](#) at the specified index.

```
IList.IsFixedSize  
IList.IsFixedSize
```

Gets a value indicating whether the [CollectionBase](#) has a fixed size.

```
IList.IsReadOnly  
IList.IsReadOnly
```

Gets a value indicating whether the [CollectionBase](#) is read-only.

```
IList.Item[Int32]  
IList.Item[Int32]
```

Gets or sets the element at the specified index.

```
IList.Remove(Object)  
IList.Remove(Object)
```

Removes the first occurrence of a specific object from the [CollectionBase](#).

## 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.

This implementation does not provide a synchronized (thread safe) wrapper for a [CollectionBase](#), but derived classes can create their own synchronized versions of the [CollectionBase](#) using the [SyncRoot](#) property.

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 causes 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.

## See Also

[ArrayList](#) [ArrayList](#)

[IList](#) [IList](#)

[ArrayList](#) [ArrayList](#)

# CollectionBase.Capacity CollectionBase.Capacity

## In this Article

Gets or sets the number of elements that the [CollectionBase](#) can contain.

```
[System.Runtime.InteropServices.ComVisible(false)]
public int Capacity { get; set; }

member this.Capacity : int with get, set
```

Returns

[Int32](#) [Int32](#)

The number of elements that the [CollectionBase](#) can contain.

Attributes

[ComVisibleAttribute](#)

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[Capacity](#) is set to a value that is less than [Count](#).

[OutOfMemoryException](#) [OutOfMemoryException](#)

There is not enough memory available on the system.

## Remarks

[Capacity](#) is the number of elements that the [CollectionBase](#) can store. [Count](#) is the number of elements that are actually in the [CollectionBase](#).

[Capacity](#) is always greater than or equal to [Count](#). If [Count](#) exceeds [Capacity](#) while adding elements, the capacity is automatically increased by reallocating the internal array before copying the old elements and adding the new elements.

The capacity can be decreased 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.

See

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

Also

# CollectionBase.Clear CollectionBase.Clear

## In this Article

Removes all objects from the [CollectionBase](#) instance. This method cannot be overridden.

```
public void Clear ();  
  
abstract member Clear : unit -> unit  
override this.Clear : unit -> unit
```

## Remarks

[Count](#) is set to zero.

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

To perform custom actions before or after the collection is cleared, override the protected [OnClear](#) or [OnClearComplete](#) method.

# CollectionBase CollectionBase

In this Article

## Overloads

<a href="#">CollectionBase()</a>	Initializes a new instance of the <a href="#">CollectionBase</a> class with the default initial capacity.
<a href="#">CollectionBase(Int32)</a>	Initializes a new instance of the <a href="#">CollectionBase</a> class with the specified capacity.

## CollectionBase()

Initializes a new instance of the [CollectionBase](#) class with the default initial capacity.

```
protected CollectionBase ();
```

### Remarks

The capacity of a [CollectionBase](#) is the number of elements that the [CollectionBase](#) can hold. As elements are added to a [CollectionBase](#), 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 [CollectionBase](#).

This constructor is an O(1) operation.

See

[Capacity](#)

Also

## CollectionBase(Int32) CollectionBase(Int32)

Initializes a new instance of the [CollectionBase](#) class with the specified capacity.

```
protected CollectionBase (int capacity);  
new System.Collections.CollectionBase : int -> System.Collections.CollectionBase
```

### Parameters

capacity

[Int32](#)

The number of elements that the new list can initially store.

### Remarks

The capacity of a [CollectionBase](#) is the number of elements that the [CollectionBase](#) can hold. As elements are added to a [CollectionBase](#), 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 [CollectionBase](#).

This constructor is an O( $n$ ) operation, where  $n$  is [capacity](#).

See

Also

Capacity

Capacity

# CollectionBase.Count CollectionBase.Count

## In this Article

Gets the number of elements contained in the [CollectionBase](#) instance. This property cannot be overridden.

```
public int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [CollectionBase](#) instance.

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

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;  
using System.Collections;  
  
public class Int16Collection : CollectionBase {  
  
    public Int16 this[ int index ] {  
        get {  
            return( (Int16) List[index] );  
        }  
        set {  
            List[index] = value;  
        }  
    }  
  
    public int Add( Int16 value ) {  
        return( List.Add( value ) );  
    }  
  
    public int IndexOf( Int16 value ) {  
        return( List.IndexOf( value ) );  
    }  
  
    public void Insert( int index, Int16 value ) {  
        List.Insert( index, value );  
    }  
  
    public void Remove( Int16 value ) {  
        List.Remove( value );  
    }  
  
    public bool Contains( Int16 value ) {  
        // If value is not of type Int16, this will return false.  
        return( List.Contains( value ) );  
    }  
  
    protected override void OnInsert( int index, Object value ) {  
        // Insert additional code to be run only when inserting values.  
    }  
  
    protected override void OnRemove( int index, Object value ) {  
        // Insert additional code to be run only when removing values.  
    }
```

```

}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Contents of the collection after removing the element 2:" );
    PrintIndexAndValues( myI16 );
}

```

```
// Uses the Count property and the Item property.  
public static void PrintIndexAndValues( Int16Collection myCol ) {  
    for ( int i = 0; i < myCol.Count; i++ )  
        Console.WriteLine( "    [{0}]: {1}", i, myCol[i] );  
    Console.WriteLine();  
}  
  
// Uses the foreach statement which hides the complexity of the enumerator.  
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.  
public static void PrintValues1( Int16Collection myCol ) {  
    foreach ( Int16 i16 in myCol )  
        Console.WriteLine( "    {0}", i16 );  
    Console.WriteLine();  
}  
  
// Uses the enumerator.  
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.  
public static void PrintValues2( Int16Collection myCol ) {  
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();  
    while ( myEnumerator.MoveNext() )  
        Console.WriteLine( "    {0}", myEnumerator.Current );  
    Console.WriteLine();  
}  
}
```

/\*  
This code produces the following output.

Contents of the collection (using foreach):

```
1  
2  
3  
5  
7
```

Contents of the collection (using enumerator):

```
1  
2  
3  
5  
7
```

Initial contents of the collection (using Count and Item):

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 5  
[4]: 7
```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13
```

```
[4]: 123  
[5]: 7
```

```
Contents of the collection after removing the element 2:
```

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

```
*/
```

# CollectionBase.GetEnumerator CollectionBase.Get Enumerator

## In this Article

Returns an enumerator that iterates through the [CollectionBase](#) instance.

```
public System.Collections.IEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the [CollectionBase](#) instance.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;  
using System.Collections;  
  
public class Int16Collection : CollectionBase {  
  
    public Int16 this[ int index ] {  
        get {  
            return( (Int16) List[index] );  
        }  
        set {  
            List[index] = value;  
        }  
    }  
  
    public int Add( Int16 value ) {  
        return( List.Add( value ) );  
    }  
  
    public int IndexOf( Int16 value ) {  
        return( List.IndexOf( value ) );  
    }  
  
    public void Insert( int index, Int16 value ) {  
        List.Insert( index, value );  
    }  
  
    public void Remove( Int16 value ) {  
        List.Remove( value );  
    }  
  
    public bool Contains( Int16 value ) {  
        // If value is not of type Int16, this will return false.  
        return( List.Contains( value ) );  
    }  
  
    protected override void OnInsert( int index, Object value ) {  
        // Insert additional code to be run only when inserting values.  
    }  
  
    protected override void OnRemove( int index, Object value ) {  
    }
```

```

// Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue )  {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value )  {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase  {

    public static void Main()  {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Contents of the collection after removing the element 2:" );
    PrintIndexAndValues( myI16 );

}

```

```
// Uses the Count property and the Item property.  
public static void PrintIndexAndValues( Int16Collection myCol ) {  
    for ( int i = 0; i < myCol.Count; i++ )  
        Console.WriteLine( " [{0}]: {1}", i, myCol[i] );  
    Console.WriteLine();  
}  
  
// Uses the foreach statement which hides the complexity of the enumerator.  
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.  
public static void PrintValues1( Int16Collection myCol ) {  
    foreach ( Int16 i16 in myCol )  
        Console.WriteLine( " {0}", i16 );  
    Console.WriteLine();  
}  
  
// Uses the enumerator.  
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.  
public static void PrintValues2( Int16Collection myCol ) {  
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();  
    while ( myEnumerator.MoveNext() )  
        Console.WriteLine( " {0}", myEnumerator.Current );  
    Console.WriteLine();  
}  
}  
  
/*  
This code produces the following output.
```

Contents of the collection (using foreach):

```
1  
2  
3  
5  
7
```

Contents of the collection (using enumerator):

```
1  
2  
3  
5  
7
```

Initial contents of the collection (using Count and Item):

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 5  
[4]: 7
```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3
```

```
[3]: 13  
[4]: 123  
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

```
*/
```

## Remarks

[Visual Basic, C#]

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, calling `Current` throws an exception. 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`, calling `Current` throws an exception. 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`. If the collection is modified between `MoveNext` and `Current`, `Current` returns the element that it is set to, even if the enumerator is already invalidated.

The enumerator does not have exclusive access to the collection; therefore, 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 causes 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.

While the `GetEnumerator` method is not visible to COM clients by default, inheriting the `CollectionBase` class can expose it and can cause undesirable behavior in COM clients.

This method is an O(1) operation.

See

[IEnumerator](#)[IEnumerator](#)

Also

# CollectionBase.ICollection.CopyTo

## In this Article

Copies the entire [CollectionBase](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

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

## Parameters

array [Array](#)

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

index [Int32](#)

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

## Exceptions

[ArgumentNullException](#)

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

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

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

[InvalidCastException](#)

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

## Remarks

The specified array must be of a compatible type.

This method uses [Array.Copy](#) to copy the elements.

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

# CollectionBase.ICollection.IsSynchronized

## In this Article

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

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

Returns

[Boolean](#)

`true` if access to the [CollectionBase](#) is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Remarks

A [CollectionBase](#) instance is not synchronized. Derived classes can provide a synchronized version of the [CollectionBase](#) using the [SyncRoot](#) property.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration:

```
// Get the ICollection interface from the CollectionBase
// derived class.
ICollection myCollection = myCollectionBase;
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

# CollectionBase.ICollection.SyncRoot

## In this Article

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

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

## Returns

[Object](#)

An object that can be used to synchronize access to the [CollectionBase](#).

## Remarks

Derived classes can provide their own synchronized version of the [CollectionBase](#) using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) of the [CollectionBase](#), not directly on the [CollectionBase](#). This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [CollectionBase](#) object.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration:

```
// Get the ICollection interface from the CollectionBase
// derived class.
ICollection myCollection = myCollectionBase;
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

# CollectionBase.IList.Add

## In this Article

Adds an object to the end of the [CollectionBase](#).

```
int IList.Add (object value);
```

### Parameters

value Object

The [Object](#) to be added to the end of the [CollectionBase](#).

### Returns

[Int32](#)

The [CollectionBase](#) index at which the `value` has been added.

### Exceptions

[NotSupportedException](#)

The [CollectionBase](#) is read-only.

-or-

The [CollectionBase](#) has a fixed size.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;
using System.Collections;

public class Int16Collection : CollectionBase {
    public Int16 this[ int index ] {
        get {
            return( (Int16) List[index] );
        }
        set {
            List[index] = value;
        }
    }

    public int Add( Int16 value ) {
        return( List.Add( value ) );
    }

    public int IndexOf( Int16 value ) {
        return( List.IndexOf( value ) );
    }

    public void Insert( int index, Int16 value ) {
        List.Insert( index, value );
    }

    public void Remove( Int16 value ) {
        List.Remove( value );
    }
}
```

```
public bool Contains( Int16 value )  {
    // If value is not of type Int16, this will return false.
    return( List.Contains( value ) );
}

protected override void OnInsert( int index, Object value )  {
    // Insert additional code to be run only when inserting values.
}

protected override void OnRemove( int index, Object value )  {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue )  {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value )  {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase  {

    public static void Main()  {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }
}
```

```

);
PrintIndexAndValues( myI16 );

// Remove an element from the collection.
myI16.Remove( (Int16) 2 );

// Display the contents of the collection using the Count property and the Item property.
Console.WriteLine( "Contents of the collection after removing the element 2:" );
PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "[{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "{0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "{0}", myEnumerator.Current );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

1
2
3
5
7

```

Contents of the collection (using enumerator):

```

1
2
3
5
7

```

Initial contents of the collection (using Count and Item):

```

[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7

```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

Contents of the collection after inserting at index 0:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 123  
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

\*/

## Remarks

If [Count](#) already equals the capacity, the capacity of the list is doubled by automatically reallocating the internal array and copying the existing elements to the new array before the new element is added.

If [Count](#) is less than the 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](#).

See

[Count](#)

Also

# CollectionBase.IList.Contains

## In this Article

Determines whether the [CollectionBase](#) contains a specific element.

```
bool IList.Contains (object value);
```

### Parameters

value [Object](#)

The [Object](#) to locate in the [CollectionBase](#).

### Returns

[Boolean](#)

`true` if the [CollectionBase](#) contains the specified `value`; otherwise, `false`.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;
using System.Collections;

public class Int16Collection : CollectionBase {
    public Int16 this[ int index ] {
        get {
            return( (Int16) List[index] );
        }
        set {
            List[index] = value;
        }
    }

    public int Add( Int16 value ) {
        return( List.Add( value ) );
    }

    public int IndexOf( Int16 value ) {
        return( List.IndexOf( value ) );
    }

    public void Insert( int index, Int16 value ) {
        List.Insert( index, value );
    }

    public void Remove( Int16 value ) {
        List.Remove( value );
    }

    public bool Contains( Int16 value ) {
        // If value is not of type Int16, this will return false.
        return( List.Contains( value ) );
    }

    protected override void OnInsert( int index, Object value ) {
        // Insert additional code to be run only when inserting values.
    }
}
```

```

protected override void OnRemove( int index, Object value ) {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

public static void Main() {

    // Create and initialize a new CollectionBase.
    Int16Collection myI16 = new Int16Collection();

    // Add elements to the collection.
    myI16.Add( (Int16) 1 );
    myI16.Add( (Int16) 2 );
    myI16.Add( (Int16) 3 );
    myI16.Add( (Int16) 5 );
    myI16.Add( (Int16) 7 );

    // Display the contents of the collection using foreach. This is the preferred method.
    Console.WriteLine( "Contents of the collection (using foreach):" );
    PrintValues1( myI16 );

    // Display the contents of the collection using the enumerator.
    Console.WriteLine( "Contents of the collection (using enumerator):" );
    PrintValues2( myI16 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
    PrintIndexAndValues( myI16 );

    // Search the collection with Contains and IndexOf.
    Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
    Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
    Console.WriteLine();

    // Insert an element into the collection at index 3.
    myI16.Insert( 3, (Int16) 13 );
    Console.WriteLine( "Contents of the collection after inserting at index 3:" );
    PrintIndexAndValues( myI16 );

    // Get and set an element using the index.
    myI16[4] = 123;
    Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
};

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Contents of the collection after removing the element 2:" );
    PrintIndexAndValues( myI16 );
}

```

```

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( " [{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( " {0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( " {0}", myEnumerator.Current );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

1
2
3
5
7

```

Contents of the collection (using enumerator):

```

1
2
3
5
7

```

Initial contents of the collection (using Count and Item):

```

[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7

```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```

[0]: 1
[1]: 2
[2]: 3
[3]: 13
[4]: 5
[5]: 7

```

Contents of the collection after setting the element at index 4 to 123:  
[0]: 1

```
[1]: 2
[2]: 3
[3]: 13
[4]: 123
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1
[1]: 3
[2]: 13
[3]: 123
[4]: 7
```

```
*/
```

## Remarks

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether [item](#) exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[Performing Culture-Insensitive String Operations](#)

Also

# CollectionBase.IList.IndexOf

## In this Article

Searches for the specified [Object](#) and returns the zero-based index of the first occurrence within the entire [CollectionBase](#).

```
int IList.IndexOf (object value);
```

### Parameters

value [Object](#)

The [Object](#) to locate in the [CollectionBase](#).

### Returns

[Int32](#)

The zero-based index of the first occurrence of `value` within the entire [CollectionBase](#), if found; otherwise, -1.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;
using System.Collections;

public class Int16Collection : CollectionBase {
    public Int16 this[ int index ] {
        get {
            return( (Int16) List[index] );
        }
        set {
            List[index] = value;
        }
    }

    public int Add( Int16 value ) {
        return( List.Add( value ) );
    }

    public int IndexOf( Int16 value ) {
        return( List.IndexOf( value ) );
    }

    public void Insert( int index, Int16 value ) {
        List.Insert( index, value );
    }

    public void Remove( Int16 value ) {
        List.Remove( value );
    }

    public bool Contains( Int16 value ) {
        // If value is not of type Int16, this will return false.
        return( List.Contains( value ) );
    }

    protected override void OnInsert( int index, Object value ) {
        // Insert additional code to be run only when inserting values.
    }
}
```

```

}

protected override void OnRemove( int index, Object value ) {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
        PrintIndexAndValues( myI16 );

        // Remove an element from the collection.
        myI16.Remove( (Int16) 2 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Contents of the collection after removing the element 2:" );
        PrintIndexAndValues( myI16 );
    }
}

```

```
PrintIndexAndValues( myInts );
}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "[{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "{0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "{0}", myEnumerator.Current );
    Console.WriteLine();
}
}

/*
This code produces the following output.
```

Contents of the collection (using foreach):

```
1
2
3
5
7
```

Contents of the collection (using enumerator):

```
1
2
3
5
7
```

Initial contents of the collection (using Count and Item):

```
[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7
```

Contains 3: True

2 is at index 1.

Contents of the collection after inserting at index 3:

```
[0]: 1
[1]: 2
[2]: 3
[3]: 13
[4]: 5
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 123  
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

\*/

## Remarks

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

This method determines equality by calling [Object.Equals](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether `item` exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

[Performing Culture-Insensitive String Operations](#)

Also

# CollectionBase.IList.Insert

## In this Article

Inserts an element into the [CollectionBase](#) at the specified index.

```
void IList.Insert (int index, object value);
```

### Parameters

index Int32

The zero-based index at which `value` should be inserted.

value Object

The [Object](#) to insert.

### Exceptions

[ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` is greater than `Count`.

[NotSupportedException](#)

The [CollectionBase](#) is read-only.

-or-

The [CollectionBase](#) has a fixed size.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;
using System.Collections;

public class Int16Collection : CollectionBase {
    public Int16 this[ int index ] {
        get {
            return( (Int16) List[index] );
        }
        set {
            List[index] = value;
        }
    }

    public int Add( Int16 value ) {
        return( List.Add( value ) );
    }

    public int IndexOf( Int16 value ) {
        return( List.IndexOf( value ) );
    }
}
```

```

public void Insert( int index, Int16 value )  {
    List.Insert( index, value );
}

public void Remove( Int16 value )  {
    List.Remove( value );
}

public bool Contains( Int16 value )  {
    // If value is not of type Int16, this will return false.
    return( List.Contains( value ) );
}

protected override void OnInsert( int index, Object value )  {
    // Insert additional code to be run only when inserting values.
}

protected override void OnRemove( int index, Object value )  {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue )  {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value )  {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase  {

    public static void Main()  {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
    }
}

```

```

myI16.Insert( 3, (Int16) 13 );
Console.WriteLine( "Contents of the collection after inserting at index 3:" );
PrintIndexAndValues( myI16 );

// Get and set an element using the index.
myI16[4] = 123;
Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
);
PrintIndexAndValues( myI16 );

// Remove an element from the collection.
myI16.Remove( (Int16) 2 );

// Display the contents of the collection using the Count property and the Item property.
Console.WriteLine( "Contents of the collection after removing the element 2:" );
PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "[{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "{0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "{0}", myEnumerator.Current );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

1
2
3
5
7

```

Contents of the collection (using enumerator):

```

1
2
3
5
7

```

Initial contents of the collection (using Count and Item):

```

[0]: 1
[1]: 2
[2]: 3

```

```
[3]: 5  
[4]: 7
```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 123  
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

```
*/
```

## Remarks

If `Count` already equals the capacity, the capacity of the list is doubled by automatically reallocating the internal array before the new element is inserted.

If `index` is equal to `Count`, `value` is added to the end of `CollectionBase`.

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.

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

# CollectionBase.IList.IsFixedSize

## In this Article

Gets a value indicating whether the [CollectionBase](#) has a fixed size.

```
bool System.Collections.IList.IsFixedSize { get; }
```

Returns

[Boolean](#)

`true` if the [CollectionBase](#) has a fixed size; otherwise, `false`. The default is `false`.

## 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.

# CollectionBase.IList.IsReadOnly

## In this Article

Gets a value indicating whether the [CollectionBase](#) is read-only.

```
bool System.Collections.IList.IsReadOnly { get; }
```

Returns

[Boolean](#)

`true` if the [CollectionBase](#) is read-only; otherwise, `false`. The default is `false`.

## 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.

# CollectionBase.IList.Item[Int32]

## In this Article

Gets or sets the element at the specified index.

```
object System.Collections.IList.Item[int index] { get; set; }
```

### Parameters

index Int32

The zero-based index of the element to get or set.

### Returns

[Object](#)

The element at the specified index.

### Exceptions

[ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` is equal to or greater than `Count`.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;
using System.Collections;

public class Int16Collection : CollectionBase {

    public Int16 this[ int index ] {
        get {
            return( (Int16) List[index] );
        }
        set {
            List[index] = value;
        }
    }

    public int Add( Int16 value ) {
        return( List.Add( value ) );
    }

    public int IndexOf( Int16 value ) {
        return( List.IndexOf( value ) );
    }

    public void Insert( int index, Int16 value ) {
        List.Insert( index, value );
    }

    public void Remove( Int16 value ) {
        List.Remove( value );
    }
}
```

```
public bool Contains( Int16 value ) {
    // If value is not of type Int16, this will return false.
    return( List.Contains( value ) );
}

protected override void OnInsert( int index, Object value ) {
    // Insert additional code to be run only when inserting values.
}

protected override void OnRemove( int index, Object value ) {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }
}
```

```

);
PrintIndexAndValues( myI16 );

// Remove an element from the collection.
myI16.Remove( (Int16) 2 );

// Display the contents of the collection using the Count property and the Item property.
Console.WriteLine( "Contents of the collection after removing the element 2:" );
PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "    [{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "    {0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "    {0}", myEnumerator.Current );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

1
2
3
5
7

```

Contents of the collection (using enumerator):

```

1
2
3
5
7

```

Initial contents of the collection (using Count and Item):

```

[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7

```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3.

Contents of the collection after inserting at index 3:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 123  
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

\*/

## Remarks

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.

See

Count

Also

# CollectionBase.IList.Remove

## In this Article

Removes the first occurrence of a specific object from the [CollectionBase](#).

```
void IList.Remove (object value);
```

### Parameters

value [Object](#)

The [Object](#) to remove from the [CollectionBase](#).

### Exceptions

[ArgumentException](#)

The `value` parameter was not found in the [CollectionBase](#) object.

[NotSupportedException](#)

The [CollectionBase](#) is read-only.

-or-

The [CollectionBase](#) has a fixed size.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;
using System.Collections;

public class Int16Collection : CollectionBase {
    public Int16 this[ int index ] {
        get {
            return( (Int16) List[index] );
        }
        set {
            List[index] = value;
        }
    }

    public int Add( Int16 value ) {
        return( List.Add( value ) );
    }

    public int IndexOf( Int16 value ) {
        return( List.IndexOf( value ) );
    }

    public void Insert( int index, Int16 value ) {
        List.Insert( index, value );
    }

    public void Remove( Int16 value ) {
        List.Remove( value );
    }
}
```

```

public bool Contains( Int16 value )  {
    // If value is not of type Int16, this will return false.
    return( List.Contains( value ) );
}

protected override void OnInsert( int index, Object value )  {
    // Insert additional code to be run only when inserting values.
}

protected override void OnRemove( int index, Object value )  {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue )  {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value )  {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase  {

    public static void Main()  {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );
}

```

```

PrintIndexAndValues( myI16 ),

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Contents of the collection after removing the element 2:" );
    PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "[{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "{0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "{0}", myEnumerator.Current );
    Console.WriteLine();
}
}

/*
This code produces the following output.

```

Contents of the collection (using foreach):

```

1
2
3
5
7

```

Contents of the collection (using enumerator):

```

1
2
3
5
7

```

Initial contents of the collection (using Count and Item):

```

[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7

```

Contains 3: True

2 is at index 1.

Contents of the collection after inserting at index 3:

```

[0]: 1

```

```
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 123  
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

\*/

## Remarks

If the [CollectionBase](#) does not contain the specified object, the [CollectionBase](#) remains unchanged. No exception is thrown.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

This method determines equality by calling [Object.Equals](#).

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.

See

[RemoveAt\(Int32\)](#)

Also

[Performing Culture-Insensitive String Operations](#)

# CollectionBase.InnerList CollectionBase.InnerList

## In this Article

Gets an [ArrayList](#) containing the list of elements in the [CollectionBase](#) instance.

```
protected System.Collections.ArrayList InnerList { get; }  
member this.InnerList : System.Collections.ArrayList
```

Returns

[ArrayList](#) [ArrayList](#)

An [ArrayList](#) representing the [CollectionBase](#) instance itself.

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

## Remarks

The On\* methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

# CollectionBase.List CollectionBase.List

## In this Article

Gets an [IList](#) containing the list of elements in the [CollectionBase](#) instance.

```
protected System.Collections.IList List { get; }  
member this.List : System.Collections.IList
```

Returns

[IList](#) [IList](#)

An [IList](#) representing the [CollectionBase](#) instance itself.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;  
using System.Collections;  
  
public class Int16Collection : CollectionBase {  
  
    public Int16 this[ int index ] {  
        get {  
            return( (Int16) List[index] );  
        }  
        set {  
            List[index] = value;  
        }  
    }  
  
    public int Add( Int16 value ) {  
        return( List.Add( value ) );  
    }  
  
    public int IndexOf( Int16 value ) {  
        return( List.IndexOf( value ) );  
    }  
  
    public void Insert( int index, Int16 value ) {  
        List.Insert( index, value );  
    }  
  
    public void Remove( Int16 value ) {  
        List.Remove( value );  
    }  
  
    public bool Contains( Int16 value ) {  
        // If value is not of type Int16, this will return false.  
        return( List.Contains( value ) );  
    }  
  
    protected override void OnInsert( int index, Object value ) {  
        // Insert additional code to be run only when inserting values.  
    }  
  
    protected override void OnRemove( int index, Object value ) {  
        // Insert additional code to be run only when removing values.  
    }
```

```

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Contents of the collection after removing the element 2:" );
    PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    ...
}

```

```
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "    [{0}]: {1}", i, myCol[i] );
    Console.WriteLine();

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "    {0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "    {0}", myEnumerator.Current );
    Console.WriteLine();
}
}
```

/\*  
This code produces the following output.

Contents of the collection (using foreach):

```
1
2
3
5
7
```

Contents of the collection (using enumerator):

```
1
2
3
5
7
```

Initial contents of the collection (using Count and Item):

```
[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7
```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```
[0]: 1
[1]: 2
[2]: 3
[3]: 13
[4]: 5
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1
[1]: 2
[2]: 3
[3]: 13
[4]: 123
[5]: 7
```

```
Contents of the collection after removing the element 2:
```

```
[0]: 1
[1]: 3
[2]: 13
[3]: 123
[4]: 7
```

```
*/
```

## Remarks

The `On*` methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

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

# CollectionBase.OnClear CollectionBase.OnClear

## In this Article

Performs additional custom processes when clearing the contents of the [CollectionBase](#) instance.

```
protected virtual void OnClear ();  
  
abstract member OnClear : unit -> unit  
override this.OnClear : unit -> unit
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the collection is cleared.

The On\* methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

If the process fails, the collection reverts back to its previous state.

The default implementation of this method is an O(1) operation.

See

[OnClearComplete\(\)](#)

Also

[OnRemove\(Int32, Object\)](#)

[OnRemove\(Int32, Object\)](#)

# CollectionBase.OnClearComplete CollectionBase.OnClearComplete

## In this Article

Performs additional custom processes after clearing the contents of the [CollectionBase](#) instance.

```
protected virtual void OnClearComplete ();  
  
abstract member OnClearComplete : unit -> unit  
override this.OnClearComplete : unit -> unit
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the collection is cleared.

The `On*` methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

The default implementation of this method is an O(1) operation.

See

[OnClear\(\)OnClear\(\)](#)

Also

[OnRemoveComplete\(Int32, Object\)](#)[OnRemoveComplete\(Int32, Object\)](#)

# CollectionBase.OnInsert CollectionBase.OnInsert

## In this Article

Performs additional custom processes before inserting a new element into the [CollectionBase](#) instance.

```
protected virtual void OnInsert (int index, object value);  
  
abstract member OnInsert : int * obj -> unit  
override this.OnInsert : int * obj -> unit
```

## Parameters

index Int32 Int32

The zero-based index at which to insert `value`.

value Object Object

The new value of the element at `index`.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;  
using System.Collections;  
  
public class Int16Collection : CollectionBase {  
  
    public Int16 this[ int index ] {  
        get {  
            return( (Int16) List[index] );  
        }  
        set {  
            List[index] = value;  
        }  
    }  
  
    public int Add( Int16 value ) {  
        return( List.Add( value ) );  
    }  
  
    public int IndexOf( Int16 value ) {  
        return( List.IndexOf( value ) );  
    }  
  
    public void Insert( int index, Int16 value ) {  
        List.Insert( index, value );  
    }  
  
    public void Remove( Int16 value ) {  
        List.Remove( value );  
    }  
  
    public bool Contains( Int16 value ) {  
        // If value is not of type Int16, this will return false.  
        return( List.Contains( value ) );  
    }  
  
    protected override void OnInsert( int index, Object value ) {  
        // Insert additional code to be run only when inserting values.  
    }  
}
```

```
}

protected override void OnRemove( int index, Object value ) {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Contents of the collection after removing the element 2:" );
}
```

```
PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "[{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "{0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "{0}", myEnumerator.Current );
    Console.WriteLine();
}
```

/\*  
This code produces the following output.

Contents of the collection (using foreach):

```
1  
2  
3  
5  
7
```

Contents of the collection (using enumerator):

```
1  
2  
3  
5  
7
```

Initial contents of the collection (using Count and Item):

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 5  
[4]: 7
```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1
[1]: 2
[2]: 3
[3]: 13
[4]: 123
[5]: 7

Contents of the collection after removing the element 2:
[0]: 1
[1]: 3
[2]: 13
[3]: 123
[4]: 7

*/
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the specified element is inserted.

The On\* methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

If the process fails, the collection reverts back to its previous state.

The default implementation of this method is an O(1) operation.

See

[OnInsertComplete\(Int32, Object\)](#)

Also

[OnSet\(Int32, Object, Object\)](#)

[OnValidate\(Object\)](#)

# CollectionBase.OnInsertComplete CollectionBase.OnInsertComplete

## In this Article

Performs additional custom processes after inserting a new element into the [CollectionBase](#) instance.

```
protected virtual void OnInsertComplete (int index, object value);  
abstract member OnInsertComplete : int * obj -> unit  
override this.OnInsertComplete : int * obj -> unit
```

## Parameters

index	<a href="#">Int32</a>
The zero-based index at which to insert <a href="#">value</a> .	
value	<a href="#">Object</a>
The new value of the element at <a href="#">index</a> .	

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the specified element is inserted.

The On\* methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

The collection reverts back to its previous state if one of the following occurs:

- The process fails.
- This method is overridden to throw an exception.

The default implementation of this method is an O(1) operation.

See

[OnInsert\(Int32, Object\)](#)

Also

[OnInsertComplete\(Int32, Object\)](#)

[OnSetComplete\(Int32, Object\)](#)

# CollectionBase.OnRemove CollectionBase.OnRemove

## In this Article

Performs additional custom processes when removing an element from the [CollectionBase](#) instance.

```
protected virtual void OnRemove (int index, object value);  
  
abstract member OnRemove : int * obj -> unit  
override this.OnRemove : int * obj -> unit
```

## Parameters

index Int32 Int32

The zero-based index at which `value` can be found.

value Object Object

The value of the element to remove from `index`.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;  
using System.Collections;  
  
public class Int16Collection : CollectionBase {  
  
    public Int16 this[ int index ] {  
        get {  
            return( (Int16) List[index] );  
        }  
        set {  
            List[index] = value;  
        }  
    }  
  
    public int Add( Int16 value ) {  
        return( List.Add( value ) );  
    }  
  
    public int IndexOf( Int16 value ) {  
        return( List.IndexOf( value ) );  
    }  
  
    public void Insert( int index, Int16 value ) {  
        List.Insert( index, value );  
    }  
  
    public void Remove( Int16 value ) {  
        List.Remove( value );  
    }  
  
    public bool Contains( Int16 value ) {  
        // If value is not of type Int16, this will return false.  
        return( List.Contains( value ) );  
    }  
  
    protected override void OnInsert( int index, Object value ) {  
        // Insert additional code to be run only when inserting values.  
    }  
}
```

```
}

protected override void OnRemove( int index, Object value ) {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
    Console.WriteLine( "Contents of the collection after removing the element 2:" );
}
```

```
PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "[{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "{0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "{0}", myEnumerator.Current );
    Console.WriteLine();
}
```

/\*  
This code produces the following output.

Contents of the collection (using foreach):

```
1  
2  
3  
5  
7
```

Contents of the collection (using enumerator):

```
1  
2  
3  
5  
7
```

Initial contents of the collection (using Count and Item):

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 5  
[4]: 7
```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1
[1]: 2
[2]: 3
[3]: 13
[4]: 123
[5]: 7

Contents of the collection after removing the element 2:
[0]: 1
[1]: 3
[2]: 13
[3]: 123
[4]: 7

*/
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the specified element is removed.

The On\* methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

If the process fails, the collection reverts back to its previous state.

The default implementation of this method is an O(1) operation.

See

[OnRemoveComplete\(Int32, Object\)](#)

Also

[OnClear\(\)](#)

# CollectionBase.OnRemoveComplete CollectionBase.OnRemoveComplete

## In this Article

Performs additional custom processes after removing an element from the [CollectionBase](#) instance.

```
protected virtual void OnRemoveComplete (int index, object value);  
abstract member OnRemoveComplete : int * obj -> unit  
override this.OnRemoveComplete : int * obj -> unit
```

## Parameters

index	<a href="#">Int32</a>
The zero-based index at which <code>value</code> can be found.	
value	<a href="#">Object</a>
The value of the element to remove from <code>index</code> .	

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the specified element is removed.

The `On*` methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

The collection reverts back to its previous state if one of the following occurs:

- The process fails.
- This method is overridden to throw an exception.

The default implementation of this method is an O(1) operation.

See

[OnRemove\(Int32, Object\)](#)

Also

[OnRemove\(Int32, Object\)](#)

[OnClearComplete\(\)](#)

# CollectionBase.OnSet CollectionBase.OnSet

## In this Article

Performs additional custom processes before setting a value in the [CollectionBase](#) instance.

```
protected virtual void OnSet (int index, object oldValue, object newValue);  
  
abstract member OnSet : int * obj * obj -> unit  
override this.OnSet : int * obj * obj -> unit
```

## Parameters

index [Int32](#) [Int32](#)

The zero-based index at which `oldValue` can be found.

oldValue [Object](#) [Object](#)

The value to replace with `newValue`.

newValue [Object](#) [Object](#)

The new value of the element at `index`.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;  
using System.Collections;  
  
public class Int16Collection : CollectionBase {  
  
    public Int16 this[ int index ] {  
        get {  
            return( (Int16) List[index] );  
        }  
        set {  
            List[index] = value;  
        }  
    }  
  
    public int Add( Int16 value ) {  
        return( List.Add( value ) );  
    }  
  
    public int IndexOf( Int16 value ) {  
        return( List.IndexOf( value ) );  
    }  
  
    public void Insert( int index, Int16 value ) {  
        List.Insert( index, value );  
    }  
  
    public void Remove( Int16 value ) {  
        List.Remove( value );  
    }  
  
    public bool Contains( Int16 value ) {  
        // If value is not of type Int16, this will return false.  
        return( List.Contains( value ) );  
    }  
}
```

```
}

protected override void OnInsert( int index, Object value ) {
    // Insert additional code to be run only when inserting values.
}

protected override void OnRemove( int index, Object value ) {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
}
```

```

myI16.Remove( (Int16) 2 );

// Display the contents of the collection using the Count property and the Item property.
Console.WriteLine( "Contents of the collection after removing the element 2:" );
PrintIndexAndValues( myI16 );

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "[{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "{0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "{0}", myEnumerator.Current );
    Console.WriteLine();
}
}

/*
This code produces the following output.

```

Contents of the collection (using foreach):

```

1
2
3
5
7

```

Contents of the collection (using enumerator):

```

1
2
3
5
7

```

Initial contents of the collection (using Count and Item):

```

[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7

```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```

[0]: 1
[1]: 2
[2]: 3
[3]: 12

```

```
[0]: 1  
[4]: 5  
[5]: 7
```

Contents of the collection after setting the element at index 4 to 123:

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 123  
[5]: 7
```

Contents of the collection after removing the element 2:

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

\*/

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the specified element is set.

The On\* methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

If the process fails, the collection reverts back to its previous state.

The default implementation of this method is an O(1) operation.

See

[OnSetComplete\(Int32, Object, Object\)](#)[OnSetComplete\(Int32, Object, Object\)](#)

Also

[OnInsert\(Int32, Object\)](#)[OnInsert\(Int32, Object\)](#)

[OnValidate\(Object\)](#)[OnValidate\(Object\)](#)

# CollectionBase.OnSetComplete CollectionBase.OnSetComplete

## In this Article

Performs additional custom processes after setting a value in the [CollectionBase](#) instance.

```
protected virtual void OnSetComplete (int index, object oldValue, object newValue);  
abstract member OnSetComplete : int * obj * obj -> unit  
override this.OnSetComplete : int * obj * obj -> unit
```

## Parameters

index [Int32](#) [Int32](#)

The zero-based index at which `oldValue` can be found.

oldValue [Object](#) [Object](#)

The value to replace with `newValue`.

newValue [Object](#) [Object](#)

The new value of the element at `index`.

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the specified element is set.

The `On*` methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

The collection reverts back to its previous state if one of the following occurs:

- The process fails.
- This method is overridden to throw an exception.

The default implementation of this method is an O(1) operation.

See

[OnSet\(Int32, Object, Object\)](#)[OnSet\(Int32, Object, Object\)](#)

Also

[OnInsertComplete\(Int32, Object\)](#)[OnInsertComplete\(Int32, Object\)](#)

# CollectionBase.OnValidate CollectionBase.OnValidate

## In this Article

Performs additional custom processes when validating a value.

```
protected virtual void OnValidate (object value);  
  
abstract member OnValidate : obj -> unit  
override this.OnValidate : obj -> unit
```

## Parameters

value Object Object

The object to validate.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`value` is `null`.

## Examples

The following code example implements the [CollectionBase](#) class and uses that implementation to create a collection of [Int16](#) objects.

```
using System;  
using System.Collections;  
  
public class Int16Collection : CollectionBase {  
  
    public Int16 this[ int index ] {  
        get {  
            return( (Int16) List[index] );  
        }  
        set {  
            List[index] = value;  
        }  
    }  
  
    public int Add( Int16 value ) {  
        return( List.Add( value ) );  
    }  
  
    public int IndexOf( Int16 value ) {  
        return( List.IndexOf( value ) );  
    }  
  
    public void Insert( int index, Int16 value ) {  
        List.Insert( index, value );  
    }  
  
    public void Remove( Int16 value ) {  
        List.Remove( value );  
    }  
  
    public bool Contains( Int16 value ) {  
        // If value is not of type Int16, this will return false.  
        return( List.Contains( value ) );  
    }  
  
    protected override void OnInsert( int index, Object value ) {
```

```
// Insert additional code to be run only when inserting values.
}

protected override void OnRemove( int index, Object value ) {
    // Insert additional code to be run only when removing values.
}

protected override void OnSet( int index, Object oldValue, Object newValue ) {
    // Insert additional code to be run only when setting values.
}

protected override void OnValidate( Object value ) {
    if ( value.GetType() != typeof(System.Int16) )
        throw new ArgumentException( "value must be of type Int16.", "value" );
}

}

public class SamplesCollectionBase {

    public static void Main() {

        // Create and initialize a new CollectionBase.
        Int16Collection myI16 = new Int16Collection();

        // Add elements to the collection.
        myI16.Add( (Int16) 1 );
        myI16.Add( (Int16) 2 );
        myI16.Add( (Int16) 3 );
        myI16.Add( (Int16) 5 );
        myI16.Add( (Int16) 7 );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintValues1( myI16 );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintValues2( myI16 );

        // Display the contents of the collection using the Count property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Count and Item):" );
        PrintIndexAndValues( myI16 );

        // Search the collection with Contains and IndexOf.
        Console.WriteLine( "Contains 3: {0}", myI16.Contains( 3 ) );
        Console.WriteLine( "2 is at index {0}.", myI16.IndexOf( 2 ) );
        Console.WriteLine();

        // Insert an element into the collection at index 3.
        myI16.Insert( 3, (Int16) 13 );
        Console.WriteLine( "Contents of the collection after inserting at index 3:" );
        PrintIndexAndValues( myI16 );

        // Get and set an element using the index.
        myI16[4] = 123;
        Console.WriteLine( "Contents of the collection after setting the element at index 4 to 123:" );
    }

    PrintIndexAndValues( myI16 );

    // Remove an element from the collection.
    myI16.Remove( (Int16) 2 );

    // Display the contents of the collection using the Count property and the Item property.
}
```

```

        Console.WriteLine( "Contents of the collection after removing the element 2:" );
        PrintIndexAndValues( myI16 );

    }

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( Int16Collection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( "    [{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( Int16Collection myCol ) {
    foreach ( Int16 i16 in myCol )
        Console.WriteLine( "    {0}", i16 );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( Int16Collection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( "    {0}", myEnumerator.Current );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

1
2
3
5
7

```

Contents of the collection (using enumerator):

```

1
2
3
5
7

```

Initial contents of the collection (using Count and Item):

```

[0]: 1
[1]: 2
[2]: 3
[3]: 5
[4]: 7

```

Contains 3: True  
2 is at index 1.

Contents of the collection after inserting at index 3:

```

[0]: 1
[1]: 2
[2]: 3
[3]: 13
[4]: 5
[5]: 7

```

```
Contents of the collection after setting the element at index 4 to 123:
```

```
[0]: 1  
[1]: 2  
[2]: 3  
[3]: 13  
[4]: 123  
[5]: 7
```

```
Contents of the collection after removing the element 2:
```

```
[0]: 1  
[1]: 3  
[2]: 13  
[3]: 123  
[4]: 7
```

```
*/
```

## Remarks

The default implementation of this method determines whether `value` is `null`, and, if so, throws [ArgumentNullException](#). It is intended to be overridden by a derived class to perform additional action when the specified element is validated.

The `On*` methods are invoked only on the instance returned by the [List](#) property, but not on the instance returned by the [InnerList](#) property.

The default implementation of this method is an O(1) operation.

See

[OnSet\(Int32, Object, Object\)](#)  
[OnSet\(Int32, Object, Object\)](#)

Also

[OnInsert\(Int32, Object\)](#)  
[OnInsert\(Int32, Object\)](#)

# CollectionBase.RemoveAt CollectionBase.RemoveAt

## In this Article

Removes the element at the specified index of the [CollectionBase](#) instance. This method is not overridable.

```
public void RemoveAt (int index);  
  
abstract member RemoveAt : int -> unit  
override this.RemoveAt : int -> unit
```

## Parameters

index	<a href="#">Int32</a>	<a href="#">Int32</a>
-------	-----------------------	-----------------------

The zero-based index of the element to remove.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

-or-

`index` is equal to or greater than `Count`.

## 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.

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

# Comparer Class

Compares two objects for equivalence, where string comparisons are case-sensitive.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public sealed class Comparer : System.Collections.IComparer,
System.Runtime.Serialization.ISerializable

type Comparer = class
    interface IComparer
    interface ISerializable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

This class is the default implementation of the [IComparer](#) interface. The [CaseInsensitiveComparer](#) class is the implementation of the [IComparer](#) interface that performs case-insensitive string comparisons. [System.Collections.Generic.Comparer<T>](#) is the generic equivalent of this class.

Comparison procedures use the [Thread.CurrentCulture](#) of the current thread unless otherwise specified. String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

## Constructors

[Comparer\(CultureInfo\)](#)

[Comparer\(CultureInfo\)](#)

Initializes a new instance of the [Comparer](#) class using the specified [CultureInfo](#).

## Fields

[Default](#)

[Default](#)

Represents an instance of [Comparer](#) that is associated with the [CurrentCulture](#) of the current thread. This field is read-only.

[DefaultInvariant](#)

[DefaultInvariant](#)

Represents an instance of [Comparer](#) that is associated with [InvariantCulture](#). This field is read-only.

## Methods

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

`Compare(Object, Object)`

Performs a case-sensitive comparison of two objects of the same type and returns a value indicating whether one is less than, equal to, or greater than the other.

`GetObjectData(SerializationInfo, StreamingContext)`

`GetObjectData(SerializationInfo, StreamingContext)`

Populates a [SerializationInfo](#) object with the data required for serialization.

## See Also

[IComparer](#) [IComparer](#)

[CultureInfo](#) [CultureInfo](#)

# Comparer.Compare

## In this Article

Performs a case-sensitive comparison of two objects of the same type and returns a value indicating whether one is less than, equal to, or greater than the other.

```
public int Compare (object a, object b);  
  
abstract member Compare : obj * obj -> int  
override this.Compare : obj * obj -> int
```

## Parameters

a Object Object

The first object to compare.

b Object Object

The second object to compare.

## Returns

[Int32](#) [Int32](#)

A signed integer that indicates the relative values of `a` and `b`, as shown in the following table.

VALUE	MEANING
Less than zero	<code>a</code> is less than <code>b</code> .
Zero	<code>a</code> equals <code>b</code> .
Greater than zero	<code>a</code> is greater than <code>b</code> .

## Exceptions

[ArgumentException](#) [ArgumentException](#)

Neither `a` nor `b` implements the [IComparable](#) interface.

-or-

`a` and `b` are of different types and neither one can handle comparisons with the other.

## Examples

The following code example shows how [Compare](#) returns different values depending on the culture associated with the [Comparer](#).

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesComparer {
    public static void Main() {
        // Creates the strings to compare.
        String str1 = "llegar";
        String str2 = "lugar";
        Console.WriteLine( "Comparing \"{0}\" and \"{1}\" ...", str1, str2 );

        // Uses the DefaultInvariant Comparer.
        Console.WriteLine( "    Invariant Comparer: {0}", Comparer.DefaultInvariant.Compare( str1, str2 ) );

        // Uses the Comparer based on the culture "es-ES" (Spanish - Spain, international sort).
        Comparer myCompIntl = new Comparer( new CultureInfo( "es-ES", false ) );
        Console.WriteLine( "    International Sort: {0}", myCompIntl.Compare( str1, str2 ) );

        // Uses the Comparer based on the culture identifier 0x040A (Spanish - Spain, traditional sort).
        Comparer myCompTrad = new Comparer( new CultureInfo( 0x040A, false ) );
        Console.WriteLine( "    Traditional Sort : {0}", myCompTrad.Compare( str1, str2 ) );
    }
}

/*
This code produces the following output.

Comparing "llegar" and "lugar" ...
    Invariant Comparer: -1
    International Sort: -1
    Traditional Sort : 1
*/

```

## Remarks

If `a` implements [IComparable](#), then `a.CompareTo(b)` is returned; otherwise, if `b` implements [IComparable](#), then the negated result of `b.CompareTo(a)` is returned.

Comparing `null` with any type is allowed and does not generate an exception when using [IComparable](#). When sorting, `null` is considered to be less than any other object.

String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

See

[CultureInfo](#)[CultureInfo](#)

Also

# Comparer

## In this Article

Initializes a new instance of the [Comparer](#) class using the specified [CultureInfo](#).

```
public Comparer (System.Globalization.CultureInfo culture);  
new System.Collections.Comparer : System.Globalization.CultureInfo -> System.Collections.Comparer
```

### Parameters

culture	<a href="#">CultureInfo</a>
---------	-----------------------------

The [CultureInfo](#) to use for the new [Comparer](#).

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`culture` is `null`.

## Examples

The following code example shows how [Compare](#) returns different values depending on the culture associated with the [Comparer](#).

```

using System;
using System.Collections;
using System.Globalization;

public class SamplesComparer {
    public static void Main() {
        // Creates the strings to compare.
        String str1 = "llegar";
        String str2 = "lugar";
        Console.WriteLine( "Comparing \"{0}\" and \"{1}\" ...", str1, str2 );

        // Uses the DefaultInvariant Comparer.
        Console.WriteLine( "    Invariant Comparer: {0}", Comparer.DefaultInvariant.Compare( str1, str2 ) );

        // Uses the Comparer based on the culture "es-ES" (Spanish - Spain, international sort).
        Comparer myCompIntl = new Comparer( new CultureInfo( "es-ES", false ) );
        Console.WriteLine( "    International Sort: {0}", myCompIntl.Compare( str1, str2 ) );

        // Uses the Comparer based on the culture identifier 0x040A (Spanish - Spain, traditional sort).
        Comparer myCompTrad = new Comparer( new CultureInfo( 0x040A, false ) );
        Console.WriteLine( "    Traditional Sort : {0}", myCompTrad.Compare( str1, str2 ) );
    }
}

/*
This code produces the following output.

Comparing "llegar" and "lugar" ...
    Invariant Comparer: -1
    International Sort: -1
    Traditional Sort : 1
*/

```

## Remarks

Comparison procedures use the specified [System.Globalization.CultureInfo](#) to determine the sort order and casing rules. String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

See

[CultureInfo](#)[CultureInfo](#)

Also

# Comparer.Default

## In this Article

Represents an instance of [Comparer](#) that is associated with the [CurrentCulture](#) of the current thread. This field is read-only.

```
public static readonly System.Collections.Comparer Default;  
staticval mutable Default : System.Collections.Comparer
```

## Returns

[Comparer](#)

## Remarks

Comparison procedures use the [Thread.CurrentCulture](#) of the current thread to determine the sort order and casing rules. String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

# Comparer.DefaultInvariant

## In this Article

Represents an instance of [Comparer](#) that is associated with [InvariantCulture](#). This field is read-only.

```
public static readonly System.Collections.Comparer DefaultInvariant;
static val mutable DefaultInvariant : System.Collections.Comparer
```

Returns

[Comparer](#) [Comparer](#)

## Examples

The following code example shows how [Compare](#) returns different values depending on the culture associated with the [Comparer](#).

```
using System;
using System.Collections;
using System.Globalization;

public class SamplesComparer {
    public static void Main() {
        // Creates the strings to compare.
        String str1 = "llegar";
        String str2 = "lugar";
        Console.WriteLine( "Comparing \"{0}\" and \"{1}\" ...", str1, str2 );

        // Uses the DefaultInvariant Comparer.
        Console.WriteLine( "    Invariant Comparer: {0}", Comparer.DefaultInvariant.Compare( str1, str2 ) );

        // Uses the Comparer based on the culture "es-ES" (Spanish - Spain, international sort).
        Comparer myCompIntl = new Comparer( new CultureInfo( "es-ES", false ) );
        Console.WriteLine( "    International Sort: {0}", myCompIntl.Compare( str1, str2 ) );

        // Uses the Comparer based on the culture identifier 0x040A (Spanish - Spain, traditional sort).
        Comparer myCompTrad = new Comparer( new CultureInfo( 0x040A, false ) );
        Console.WriteLine( "    Traditional Sort : {0}", myCompTrad.Compare( str1, str2 ) );
    }
}

/*
This code produces the following output.

Comparing "llegar" and "lugar" ...
Invariant Comparer: -1
International Sort: -1
Traditional Sort : 1
*/
```

## Remarks

Comparison procedures use the [CultureInfo.InvariantCulture](#) to determine the sort order and casing rules. String comparisons might have different results depending on the culture. For more information on culture-specific comparisons, see the [System.Globalization](#) namespace and [Globalization and Localization](#).

# Comparer.GetObjectData Comparer.GetObjectData

## In this Article

Populates a [SerializationInfo](#) object with the data required for serialization.

```
[System.Security.SecurityCritical]
public void GetObjectData (System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);

abstract member GetObjectData : System.Runtime.Serialization.SerializationInfo *
System.Runtime.Serialization.StreamingContext -> unit
override this.GetObjectData : System.Runtime.Serialization.SerializationInfo *
System.Runtime.Serialization.StreamingContext -> unit
```

## Parameters

info [SerializationInfo](#) [SerializationInfo](#)

The object to populate with data.

context [StreamingContext](#) [StreamingContext](#)

The context information about the source or destination of the serialization.

Attributes [SecurityCriticalAttribute](#)

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`info` is `null`.

## Remarks

While the [GetObjectData](#) method is not visible to COM clients by default, inheriting the [Comparer](#) class can expose it and can cause undesirable behavior in COM clients.

# DictionaryBase DictionaryBase Class

Provides the `abstract` base class for a strongly typed collection of key/value pairs.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public abstract class DictionaryBase : System.Collections.IDictionary

type DictionaryBase = class
    interface IDictionary
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

`Object` `Object`

## Remarks

### Important

We don't recommend that you use the `DictionaryBase` class for new development. Instead, we recommend that you use the generic `Dictionary< TKey, TValue >` or `KeyedCollection< TKey, TItem >` class . For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

The C# `foreach` statement and the Visual Basic `For Each` statement return an object of the type of the elements in the collection. Since each element of the `DictionaryBase` is a key/value pair, the element type is not the type of the key or the type of the value. Instead, the element type is `DictionaryEntry`.

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.

## Constructors

`DictionaryBase()`

`DictionaryBase()`

Initializes a new instance of the `DictionaryBase` class.

## Properties

`Count`

`Count`

Gets the number of elements contained in the `DictionaryBase` instance.

`Dictionary`

`Dictionary`

Gets the list of elements contained in the [DictionaryBase](#) instance.

InnerHashtable

InnerHashtable

Gets the list of elements contained in the [DictionaryBase](#) instance.

## Methods

Clear()

Clear()

Clears the contents of the [DictionaryBase](#) instance.

CopyTo(Array, Int32)

CopyTo(Array, Int32)

Copies the [DictionaryBase](#) elements to a one-dimensional [Array](#) at the specified index.

GetEnumerator()

GetEnumerator()

Returns an [IDictionaryEnumerator](#) that iterates through the [DictionaryBase](#) instance.

OnClear()

OnClear()

Performs additional custom processes before clearing the contents of the [DictionaryBase](#) instance.

OnClearComplete()

OnClearComplete()

Performs additional custom processes after clearing the contents of the [DictionaryBase](#) instance.

OnGet(Object, Object)

OnGet(Object, Object)

Gets the element with the specified key and value in the [DictionaryBase](#) instance.

OnInsert(Object, Object)

OnInsert(Object, Object)

Performs additional custom processes before inserting a new element into the [DictionaryBase](#) instance.

```
OnInsertComplete(Object, Object)  
OnInsertComplete(Object, Object)
```

Performs additional custom processes after inserting a new element into the [DictionaryBase](#) instance.

```
OnRemove(Object, Object)  
OnRemove(Object, Object)
```

Performs additional custom processes before removing an element from the [DictionaryBase](#) instance.

```
OnRemoveComplete(Object, Object)  
OnRemoveComplete(Object, Object)
```

Performs additional custom processes after removing an element from the [DictionaryBase](#) instance.

```
OnSet(Object, Object, Object)  
OnSet(Object, Object, Object)
```

Performs additional custom processes before setting a value in the [DictionaryBase](#) instance.

```
OnSetComplete(Object, Object, Object)  
OnSetComplete(Object, Object, Object)
```

Performs additional custom processes after setting a value in the [DictionaryBase](#) instance.

```
OnValidate(Object, Object)  
OnValidate(Object, Object)
```

Performs additional custom processes when validating the element with the specified key and value.

```
ICollection.IsSynchronized  
ICollection.IsSynchronized
```

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

```
ICollection.SyncRoot  
ICollection.SyncRoot
```

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

```
IDictionary.Add(Object, Object)  
IDictionary.Add(Object, Object)
```

Adds an element with the specified key and value into the [DictionaryBase](#).

```
IDictionary.Contains(Object)  
IDictionary.Contains(Object)
```

Determines whether the [DictionaryBase](#) contains a specific key.

```
IDictionary.IsFixedSize  
IDictionary.IsFixedSize
```

Gets a value indicating whether a [DictionaryBase](#) object has a fixed size.

```
IDictionary.IsReadOnly  
IDictionary.IsReadOnly
```

Gets a value indicating whether a [DictionaryBase](#) object is read-only.

```
IDictionary.Item[Object]  
IDictionary.Item[Object]
```

Gets or sets the value associated with the specified key.

```
IDictionary.Keys  
IDictionary.Keys
```

Gets an [ICollection](#) object containing the keys in the [DictionaryBase](#) object.

```
IDictionary.Remove(Object)  
IDictionary.Remove(Object)
```

Removes the element with the specified key from the [DictionaryBase](#).

```
IDictionary.Values  
IDictionary.Values
```

Gets an [ICollection](#) object containing the values in the [DictionaryBase](#) object.

```
IEnumerable.GetEnumerator()  
IEnumerable.GetEnumerator()
```

Returns an [IEnumerator](#) that iterates through the [DictionaryBase](#).

## 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.

This implementation does not provide a synchronized (thread-safe) wrapper for a [DictionaryBase](#), but derived classes can create their own synchronized versions of the [DictionaryBase](#) using the [SyncRoot](#) property.

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 causes 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.

## See Also

[Hashtable](#) [Hashtable](#)

[IDictionary](#) [IDictionary](#)

[Hashtable](#) [Hashtable](#)

# DictionaryBase.Clear DictionaryBase.Clear

## In this Article

Clears the contents of the [DictionaryBase](#) instance.

```
public void Clear ();  
  
abstract member Clear : unit -> unit  
override this.Clear : unit -> unit
```

## Remarks

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

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

# DictionaryBase.CopyTo DictionaryBase.CopyTo

## In this Article

Copies the [DictionaryBase](#) elements to a one-dimensional [Array](#) at the specified index.

```
public void CopyTo (Array array, int index);  
  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

## Parameters

array [Array](#) [Array](#)

The one-dimensional [Array](#) that is the destination of the [DictionaryEntry](#) objects copied from the [DictionaryBase](#) instance. The [Array](#) must have zero-based indexing.

index [Int32](#) [Int32](#)

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

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#) [ArgumentException](#)

[array](#) is multidimensional.

-or-

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

[InvalidCastException](#) [InvalidCastException](#)

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

## Remarks

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [DictionaryBase](#).

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

See

[Array](#)[Array](#)

Also

[GetEnumerator\(\)](#)[GetEnumerator\(\)](#)

# DictionaryBase.Count DictionaryBase.Count

## In this Article

Gets the number of elements contained in the [DictionaryBase](#) instance.

```
public int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [DictionaryBase](#) instance.

## Remarks

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

# DictionaryBase.Dictionary DictionaryBase.Dictionary

## In this Article

Gets the list of elements contained in the [DictionaryBase](#) instance.

```
protected System.Collections.IDictionary Dictionary { get; }  
member this.Dictionary : System.Collections.IDictionary
```

Returns

[IDictionary](#) [IDictionary](#)

An [IDictionary](#) representing the [DictionaryBase](#) instance itself.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;  
using System.Collections;  
  
public class ShortStringDictionary : DictionaryBase {  
  
    public String this[ String key ] {  
        get {  
            return( (String) Dictionary[key] );  
        }  
        set {  
            Dictionary[key] = value;  
        }  
    }  
  
    public ICollection Keys {  
        get {  
            return( Dictionary.Keys );  
        }  
    }  
  
    public ICollection Values {  
        get {  
            return( Dictionary.Values );  
        }  
    }  
  
    public void Add( String key, String value ) {  
        Dictionary.Add( key, value );  
    }  
  
    public bool Contains( String key ) {  
        return( Dictionary.Contains( key ) );  
    }  
  
    public void Remove( String key ) {  
        Dictionary.Remove( key );  
    }  
  
    protected override void OnInsert( Object key, Object value ) {  
        if ( key.GetType() != typeof(System.String) )  
            throw new ArgumentException( "key must be of type String.", "key" );  
        else {  
            String strKey = (String) key;  
        }  
    }  
}
```

```

        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
    }
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( newValue.GetType() != typeof(System.String) )
        throw new ArgumentException( "newValue must be of type String.", "newValue" );
    else {
        String strValue = (String) newValue;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "newValue must be no more than 5 characters in length.",
"newValue" );
    }
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
    }
}

```

```
}
```

```
public class SamplesDictionaryBase {

    public static void Main() {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );

        // Displays the contents of the collection.
        Console.WriteLine( "After removing \"Two\":" );
        PrintKeysAndValues1( mySSC );

    }

    // Uses the foreach statement which hides the complexity of the enumerator.
    // NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
    public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
        foreach ( DictionaryEntry mvDE in myCol )

```

```

        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        Console.WriteLine();
    }

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

System.ArgumentException: value must be no more than 5 characters in length.

Parameter name: value

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

System.ArgumentException: key must be no more than 5 characters in length.

Parameter name: key

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

Contains "Three": True

Contains "Twelve": False

```
After removing "Two":
```

```
Three : abc
Five  : abcde
One   : a
Four  : abcd
```

```
*/
```

## Remarks

The `On*` methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

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

# DictionaryBase

## In this Article

Initializes a new instance of the [DictionaryBase](#) class.

```
protected DictionaryBase ();
```

## Remarks

This constructor is an O(1) operation.

# DictionaryBase.GetEnumerator DictionaryBase.Get Enumerator

## In this Article

Returns an [IDictionaryEnumerator](#) that iterates through the [DictionaryBase](#) instance.

```
public System.Collections.IDictionaryEnumerator GetEnumerator ();  
abstract member GetEnumerator : unit -> System.Collections.IDictionaryEnumerator  
override this.GetEnumerator : unit -> System.Collections.IDictionaryEnumerator
```

Returns

[IDictionaryEnumerator](#) [IDictionaryEnumerator](#)

An [IDictionaryEnumerator](#) for the [DictionaryBase](#) instance.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;  
using System.Collections;  
  
public class ShortStringDictionary : DictionaryBase {  
  
    public String this[ String key ] {  
        get {  
            return( (String) Dictionary[key] );  
        }  
        set {  
            Dictionary[key] = value;  
        }  
    }  
  
    public ICollection Keys {  
        get {  
            return( Dictionary.Keys );  
        }  
    }  
  
    public ICollection Values {  
        get {  
            return( Dictionary.Values );  
        }  
    }  
  
    public void Add( String key, String value ) {  
        Dictionary.Add( key, value );  
    }  
  
    public bool Contains( String key ) {  
        return( Dictionary.Contains( key ) );  
    }  
  
    public void Remove( String key ) {  
        Dictionary.Remove( key );  
    }  
  
    protected override void OnInsert( Object key, Object value ) {  
        if ( key.GetType() != typeof(System.String) )
```

```

        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
    }
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( newValue.GetType() != typeof(System.String) )
        throw new ArgumentException( "newValue must be of type String.", "newValue" );
    else {
        String strValue = (String) newValue;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "newValue must be no more than 5 characters in length.",
"newValue" );
    }
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
", "
);
    }
}

```

```
"value" );
        }
    }

}

public class SamplesDictionaryBase  {

    public static void Main()  {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try  {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try  {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );

        // Displays the contents of the collection.
        Console.WriteLine( "After removing \"Two\":" );
        PrintKeysAndValues1( mySSC );

    }

    // Uses the foreach statement which hides the complexity of the enumerator.
}
```

```
// Uses the foreach statement which hides the complexity of the enumerator.
```

```
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
```

```
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
```

```
    foreach ( DictionaryEntry myDE in myCol )
```

```
        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
```

```
    Console.WriteLine();
```

```
}
```

```
// Uses the enumerator.
```

```
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
```

```
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
```

```
    DictionaryEntry myDE;
```

```
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
```

```
    while ( myEnumerator.MoveNext() )
```

```
        if ( myEnumerator.Current != null ) {
```

```
            myDE = (DictionaryEntry) myEnumerator.Current;
```

```
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
```

```
        }
```

```
    Console.WriteLine();
```

```
}
```

```
// Uses the Keys property and the Item property.
```

```
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
```

```
    ICollection myKeys = myCol.Keys;
```

```
    foreach ( String k in myKeys )
```

```
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
```

```
    Console.WriteLine();
```

```
}
```

```
}
```

```
*
```

```
This code produces the following output.
```

```
Contents of the collection (using foreach):
```

```
Three : abc
```

```
Five : abcde
```

```
Two : ab
```

```
One : a
```

```
Four : abcd
```

```
Contents of the collection (using enumerator):
```

```
Three : abc
```

```
Five : abcde
```

```
Two : ab
```

```
One : a
```

```
Four : abcd
```

```
Initial contents of the collection (using Keys and Item):
```

```
Three : abc
```

```
Five : abcde
```

```
Two : ab
```

```
One : a
```

```
Four : abcd
```

```
System.ArgumentException: value must be no more than 5 characters in length.
```

```
Parameter name: value
```

```
at ShortStringDictionary.OnValidate(Object key, Object value)
```

```
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
```

```
at SamplesDictionaryBase.Main()
```

```
System.ArgumentException: key must be no more than 5 characters in length.
```

```
Parameter name: key
```

```
at ShortStringDictionary.OnValidate(Object key, Object value)
```

```
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
```

```
at SamplesDictionaryBase.Main()
```

```
Contains "Three": True  
Contains "Twelve": False
```

```
After removing "Two":
```

```
Three : abc  
Five : abcde  
One : a  
Four : abcd
```

```
*/
```

## 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 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.

This method is an O(1) operation.

See

[IDictionaryEnumerator](#)[DictionaryEnumerator](#)

Also

[IEnumerator](#)[Enumerator](#)

# DictionaryBase.ICollection.IsSynchronized

## In this Article

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

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

Returns

[Boolean](#)

`true` if access to the [DictionaryBase](#) object is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) property during the entire enumeration.

```
ICollection myCollection = new ShortStringDictionary();
lock(myCollection.SyncRoot)
{
    foreach (Object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

## Remarks

A [DictionaryBase](#) object is not synchronized. Derived classes can provide a synchronized version of the [DictionaryBase](#) class using the [SyncRoot](#) property.

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 causes 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.

# DictionaryBase.ICollection.SyncRoot

## In this Article

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

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

Returns

[Object](#)

An object that can be used to synchronize access to the [DictionaryBase](#) object.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) property during the entire enumeration.

```
ICollection myCollection = new ShortStringDictionary();
lock(myCollection.SyncRoot)
{
    foreach (Object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

## Remarks

Derived classes can provide their own synchronized version of the [DictionaryBase](#) class using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) property of the [DictionaryBase](#) object, not directly on the [DictionaryBase](#) object. This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [DictionaryBase](#) object.

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 causes 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.

# DictionaryBase.IDictionary.Add

## In this Article

Adds an element with the specified key and value into the [DictionaryBase](#).

```
void IDictionary.Add (object key, object value);
```

### Parameters

key Object

The key of the element to add.

value Object

The value of the element to add.

### Exceptions

[ArgumentNullException](#)

key is null.

[ArgumentException](#)

An element with the same key already exists in the [DictionaryBase](#).

[NotSupportedException](#)

The [DictionaryBase](#) is read-only.

-or-

The [DictionaryBase](#) has a fixed size.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;
using System.Collections;

public class ShortStringDictionary : DictionaryBase {

    public String this[ String key ] {
        get {
            return( (String) Dictionary[key] );
        }
        set {
            Dictionary[key] = value;
        }
    }

    public ICollection Keys {
        get {
            return( Dictionary.Keys );
        }
    }

    public ICollection Values {
        get {

```

```

        return( Dictionary.Values );
    }

}

public void Add( String key, String value ) {
    Dictionary.Add( key, value );
}

public bool Contains( String key ) {
    return( Dictionary.Contains( key ) );
}

public void Remove( String key ) {
    Dictionary.Remove( key );
}

protected override void OnInsert( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( value.GetType() != typeof(System.String) )
    throw new ArgumentException( "value must be of type String.", "value" );
else {
    String strValue = (String) value;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( newValue.GetType() != typeof(System.String) )
    throw new ArgumentException( "newValue must be of type String.", "newValue" );
else {
    String strValue = (String) newValue;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
}

```

```
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
    }
}

}

public class SamplesDictionaryBase {

public static void Main() {

    // Creates and initializes a new DictionaryBase.
    ShortStringDictionary mySSC = new ShortStringDictionary();

    // Adds elements to the collection.
    mySSC.Add( "One", "a" );
    mySSC.Add( "Two", "ab" );
    mySSC.Add( "Three", "abc" );
    mySSC.Add( "Four", "abcd" );
    mySSC.Add( "Five", "abcde" );

    // Display the contents of the collection using foreach. This is the preferred method.
    Console.WriteLine( "Contents of the collection (using foreach):" );
    PrintKeysAndValues1( mySSC );

    // Display the contents of the collection using the enumerator.
    Console.WriteLine( "Contents of the collection (using enumerator):" );
    PrintKeysAndValues2( mySSC );

    // Display the contents of the collection using the Keys property and the Item property.
    Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
    PrintKeysAndValues3( mySSC );

    // Tries to add a value that is too long.
    try {
        mySSC.Add( "Ten", "abcdefghijkl" );
    }
    catch ( ArgumentException e ) {
        Console.WriteLine( e.ToString() );
    }

    // Tries to add a key that is too long.
    try {
        mySSC.Add( "Eleven", "ijk" );
    }
    catch ( ArgumentException e ) {
        Console.WriteLine( e.ToString() );
    }
}
```

```

        ,
Console.WriteLine();

// Searches the collection with Contains.
Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
Console.WriteLine();

// Removes an element from the collection.
mySSC.Remove( "Two" );

// Displays the contents of the collection.
Console.WriteLine( "After removing \"Two\":\"" );
PrintKeysAndValues1( mySSC );

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( "  {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( "  {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( "  {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}
}

/*
This code produces the following output.

```

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

```

Initial contents of the collection (using Keys and Item):
Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

System.ArgumentException: value must be no more than 5 characters in length.
Parameter name: value
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

System.ArgumentException: key must be no more than 5 characters in length.
Parameter name: key
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

Contains "Three": True
Contains "Twelve": False

After removing "Two":
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/

```

## Remarks

An object that has no correlation between its state and its hash code value should typically not be used as the key. For example, [String](#) objects are better than [StringBuilder](#) objects for use as keys.

You can also use the [Item\[Object\]](#) property to add new elements by setting the value of a key that does not exist in the [DictionaryBase](#); for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the [DictionaryBase](#), setting the [Item\[Object\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

This method is an O(1) operation.

# DictionaryBase.IDictionary.Contains

## In this Article

Determines whether the [DictionaryBase](#) contains a specific key.

```
bool IDictionary.Contains (object key);
```

### Parameters

key Object

The key to locate in the [DictionaryBase](#).

### Returns

[Boolean](#)

`true` if the [DictionaryBase](#) contains an element with the specified key; otherwise, `false`.

### Exceptions

[ArgumentNullException](#)

`key` is `null`.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;
using System.Collections;

public class ShortStringDictionary : DictionaryBase {

    public String this[ String key ] {
        get {
            return( (String) Dictionary[key] );
        }
        set {
            Dictionary[key] = value;
        }
    }

    public ICollection Keys {
        get {
            return( Dictionary.Keys );
        }
    }

    public ICollection Values {
        get {
            return( Dictionary.Values );
        }
    }

    public void Add( String key, String value ) {
        Dictionary.Add( key, value );
    }

    public bool Contains( String key ) {
        return( Dictionary.Contains( key ) );
    }
}
```

```

public void Remove( String key ) {
    Dictionary.Remove( key );
}

protected override void OnInsert( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( value.GetType() != typeof(System.String) )
    throw new ArgumentException( "value must be of type String.", "value" );
else {
    String strValue = (String) value;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
}
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( newValue.GetType() != typeof(System.String) )
    throw new ArgumentException( "newValue must be of type String.", "newValue" );
else {
    String strValue = (String) newValue;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
}
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}
}

```

```

        if ( value.GetType() != typeof(System.String) )
            throw new ArgumentException( "value must be of type String.", "value" );
        else {
            String strValue = (String) value;
            if ( strValue.Length > 5 )
                throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
        }
    }

}

public class SamplesDictionaryBase {

    public static void Main() {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );
    }
}

```

```

// Displays the contents of the collection.
Console.WriteLine( "After removing \"Two\":");
PrintKeysAndValues1( mySSC );

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

System.ArgumentException: value must be no more than 5 characters in length.  
Parameter name: value  
at ShortStringDictionary.OnValidate(Object key, Object value)

```
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()
System.ArgumentException: key must be no more than 5 characters in length.
Parameter name: key
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

Contains "Three": True
Contains "Twelve": False

After removing "Two":
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/
```

## Remarks

This method is an O(1) operation.

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `key` to determine whether `item` exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `key` parameter on the objects in the collection.

See

[Performing Culture-Insensitive String Operations](#)

Also

# DictionaryBase.IDictionary.IsFixedSize

## In this Article

Gets a value indicating whether a [DictionaryBase](#) object has a fixed size.

```
bool System.Collections.IDictionary.IsFixedSize { get; }
```

Returns

[Boolean](#)

`true` if the [DictionaryBase](#) object has a fixed size; otherwise, `false`. The default is `false`.

## Remarks

A collection with a fixed size does not allow the addition or removal of elements after the collection is created, but does allow 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.

# DictionaryBase.IDictionary.IsReadOnly

## In this Article

Gets a value indicating whether a [DictionaryBase](#) object is read-only.

```
bool System.Collections.IDictionary.IsReadOnly { get; }
```

Returns

[Boolean](#)

`true` if the [DictionaryBase](#) object is read-only; otherwise, `false`. The default is `false`.

## 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.

# DictionaryBase.IDictionary.Item[Object]

## In this Article

Gets or sets the value associated with the specified key.

```
object System.Collections.IDictionary.Item[object key] { get; set; }
```

### Parameters

key Object

The key whose value to get or set.

### Returns

[Object](#)

The value associated with the specified key. If the specified key is not found, attempting to get it returns `null`, and attempting to set it creates a new element using the specified key.

### Exceptions

[ArgumentNullException](#)

`key` is `null`.

[NotSupportedException](#)

The property is set and the [DictionaryBase](#) is read-only.

-or-

The property is set, `key` does not exist in the collection, and the [DictionaryBase](#) has a fixed size.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;
using System.Collections;

public class ShortStringDictionary : DictionaryBase {

    public String this[ String key ] {
        get {
            return( (String) Dictionary[key] );
        }
        set {
            Dictionary[key] = value;
        }
    }

    public ICollection Keys {
        get {
            return( Dictionary.Keys );
        }
    }

    public ICollection Values {
        get {
            return( Dictionary.Values );
        }
    }
}
```

```

        }

    public void Add( String key, String value ) {
        Dictionary.Add( key, value );
    }

    public bool Contains( String key ) {
        return( Dictionary.Contains( key ) );
    }

    public void Remove( String key ) {
        Dictionary.Remove( key );
    }

    protected override void OnInsert( Object key, Object value ) {
        if ( key.GetType() != typeof(System.String) )
            throw new ArgumentException( "key must be of type String.", "key" );
        else {
            String strKey = (String) key;
            if ( strKey.Length > 5 )
                throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
        }
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
    }
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( newValue.GetType() != typeof(System.String) )
    throw new ArgumentException( "newValue must be of type String.", "newValue" );
else {
    String strValue = (String) newValue;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
}
}

```

```

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( value.GetType() != typeof(System.String) )
    throw new ArgumentException( "value must be of type String.", "value" );
else {
    String strValue = (String) value;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
}
}

}

public class SamplesDictionaryBase {

public static void Main() {

    // Creates and initializes a new DictionaryBase.
    ShortStringDictionary mySSC = new ShortStringDictionary();

    // Adds elements to the collection.
    mySSC.Add( "One", "a" );
    mySSC.Add( "Two", "ab" );
    mySSC.Add( "Three", "abc" );
    mySSC.Add( "Four", "abcd" );
    mySSC.Add( "Five", "abcde" );

    // Display the contents of the collection using foreach. This is the preferred method.
    Console.WriteLine( "Contents of the collection (using foreach):" );
    PrintKeysAndValues1( mySSC );

    // Display the contents of the collection using the enumerator.
    Console.WriteLine( "Contents of the collection (using enumerator):" );
    PrintKeysAndValues2( mySSC );

    // Display the contents of the collection using the Keys property and the Item property.
    Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
    PrintKeysAndValues3( mySSC );

    // Tries to add a value that is too long.
    try {
        mySSC.Add( "Ten", "abcdefghijkl" );
    }
    catch ( ArgumentException e ) {
        Console.WriteLine( e.ToString() );
    }

    // Tries to add a key that is too long.
    try {
        mySSC.Add( "Eleven", "ijk" );
    }
    catch ( ArgumentException e ) {
        Console.WriteLine( e.ToString() );
    }
}

```

```

Console.WriteLine();

// Searches the collection with Contains.
Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
Console.WriteLine();

// Removes an element from the collection.
mySSC.Remove( "Two" );

// Displays the contents of the collection.
Console.WriteLine( "After removing \"Two\":" );
PrintKeysAndValues1( mySSC );

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( "  {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( "  {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( "  {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}
}

/*
This code produces the following output.

```

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```
INITIAL CONTENTS OF THE COLLECTION (USING Keys AND Items).
```

```
Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd
```

```
System.ArgumentException: value must be no more than 5 characters in length.
```

```
Parameter name: value
```

```
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()
```

```
System.ArgumentException: key must be no more than 5 characters in length.
```

```
Parameter name: key
```

```
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()
```

```
Contains "Three": True
```

```
Contains "Twelve": False
```

```
After removing "Two":
```

```
Three : abc
Five  : abcde
One   : a
Four  : abcd
```

```
*/
```

## Remarks

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\[Object\]](#) property to add new elements by setting the value of a key that does not exist in the [DictionaryBase](#); for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the [DictionaryBase](#), setting the [Item\[Object\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

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

# DictionaryBase.IDictionary.Keys

## In this Article

Gets an [ICollection](#) object containing the keys in the [DictionaryBase](#) object.

```
System.Collections.ICollection System.Collections.IDictionary.Keys { get; }
```

## Returns

[ICollection](#)

An [ICollection](#) object containing the keys in the [DictionaryBase](#) object.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) property of 5 characters or less.

```
using System;
using System.Collections;

public class ShortStringDictionary : DictionaryBase {

    public String this[ String key ] {
        get {
            return( (String) Dictionary[key] );
        }
        set {
            Dictionary[key] = value;
        }
    }

    public ICollection Keys {
        get {
            return( Dictionary.Keys );
        }
    }

    public ICollection Values {
        get {
            return( Dictionary.Values );
        }
    }

    public void Add( String key, String value ) {
        Dictionary.Add( key, value );
    }

    public bool Contains( String key ) {
        return( Dictionary.Contains( key ) );
    }

    public void Remove( String key ) {
        Dictionary.Remove( key );
    }

    protected override void OnInsert( Object key, Object value ) {
        if ( key.GetType() != typeof(System.String) )
            throw new ArgumentException( "key must be of type String.", "key" );
        else {
            String strKey = (String) key;
            if ( strKey.Length > 5 )
                throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
        }
    }
}
```

```

        throw new ArgumentException( "key must be no more than 5 characters in length.", key
);
}

if ( value.GetType() != typeof(System.String) )
    throw new ArgumentException( "value must be of type String.", "value" );
else {
    String strValue = (String) value;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
}
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( newValue.GetType() != typeof(System.String) )
        throw new ArgumentException( "newValue must be of type String.", "newValue" );
    else {
        String strValue = (String) newValue;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "newValue must be no more than 5 characters in length.",
"newValue" );
    }
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
    }
}
}

```

```

public class SamplesDictionaryBase {

    public static void Main() {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );

        // Displays the contents of the collection.
        Console.WriteLine( "After removing \"Two\":" );
        PrintKeysAndValues1( mySSC );

    }

    // Uses the foreach statement which hides the complexity of the enumerator.
    // NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
    public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
        foreach ( DictionaryEntry myDE in myCol )
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    }
}

```

```

        Console.WriteLine();
    }

    // Uses the enumerator.
    // NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
    public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
        DictionaryEntry myDE;
        System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
        while ( myEnumerator.MoveNext() )
            if ( myEnumerator.Current != null ) {
                myDE = (DictionaryEntry) myEnumerator.Current;
                Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
            }
        Console.WriteLine();
    }

    // Uses the Keys property and the Item property.
    public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
        ICollection myKeys = myCol.Keys;
        foreach ( String k in myKeys )
            Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
        Console.WriteLine();
    }

}

/*
This code produces the following output.

```

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

`System.ArgumentException: value must be no more than 5 characters in length.`

Parameter name: value

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

`System.ArgumentException: key must be no more than 5 characters in length.`

Parameter name: key

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

Contains "Three": True

Contains "Twelve": False

After removing "Two":

```
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/
```

## Remarks

The order of the keys in the [ICollection](#) object is unspecified, but is the same order as the associated values in the [ICollection](#) object returned by the [Values](#) property.

The returned [ICollection](#) is not a static copy; instead, the [ICollection](#) refers back to the keys in the original [DictionaryBase](#) object. Therefore, changes to the [DictionaryBase](#) continue to be reflected in the returned [ICollection](#).

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

See

[ICollection](#)

Also

# DictionaryBase.IDictionary.Remove

## In this Article

Removes the element with the specified key from the [DictionaryBase](#).

```
void IDictionary.Remove (object key);
```

### Parameters

key [Object](#)

The key of the element to remove.

### Exceptions

[ArgumentNullException](#)

`key` is `null`.

[NotSupportedException](#)

The [DictionaryBase](#) is read-only.

-or-

The [DictionaryBase](#) has a fixed size.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;
using System.Collections;

public class ShortStringDictionary : DictionaryBase {

    public String this[ String key ] {
        get {
            return( (String) Dictionary[key] );
        }
        set {
            Dictionary[key] = value;
        }
    }

    public ICollection Keys {
        get {
            return( Dictionary.Keys );
        }
    }

    public ICollection Values {
        get {
            return( Dictionary.Values );
        }
    }

    public void Add( String key, String value ) {
        Dictionary.Add( key, value );
    }
}
```

```

public bool Contains( String key ) {
    return( Dictionary.Contains( key ) );
}

public void Remove( String key ) {
    Dictionary.Remove( key );
}

protected override void OnInsert( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( value.GetType() != typeof(System.String) )
    throw new ArgumentException( "value must be of type String.", "value" );
else {
    String strValue = (String) value;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( newValue.GetType() != typeof(System.String) )
    throw new ArgumentException( "newValue must be of type String.", "newValue" );
else {
    String strValue = (String) newValue;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

```

```
        throw new ArgumentException( "key must be no more than 5 characters in length.", key
);
}

if ( value.GetType() != typeof(System.String) )
    throw new ArgumentException( "value must be of type String.", "value" );
else {
    String strValue = (String) value;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
}
}

}

public class SamplesDictionaryBase {

public static void Main() {

// Creates and initializes a new DictionaryBase.
ShortStringDictionary mySSC = new ShortStringDictionary();

// Adds elements to the collection.
mySSC.Add( "One", "a" );
mySSC.Add( "Two", "ab" );
mySSC.Add( "Three", "abc" );
mySSC.Add( "Four", "abcd" );
mySSC.Add( "Five", "abcde" );

// Display the contents of the collection using foreach. This is the preferred method.
Console.WriteLine( "Contents of the collection (using foreach):" );
PrintKeysAndValues1( mySSC );

// Display the contents of the collection using the enumerator.
Console.WriteLine( "Contents of the collection (using enumerator):" );
PrintKeysAndValues2( mySSC );

// Display the contents of the collection using the Keys property and the Item property.
Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
PrintKeysAndValues3( mySSC );

// Tries to add a value that is too long.
try {
    mySSC.Add( "Ten", "abcdefghijkl" );
}
catch ( ArgumentException e ) {
    Console.WriteLine( e.ToString() );
}

// Tries to add a key that is too long.
try {
    mySSC.Add( "Eleven", "ijk" );
}
catch ( ArgumentException e ) {
    Console.WriteLine( e.ToString() );
}

Console.WriteLine();

// Searches the collection with Contains.
Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
Console.WriteLine();
```

```

// Removes an element from the collection.
mySSC.Remove( "Two" );

// Displays the contents of the collection.
Console.WriteLine( "After removing \"Two\":" );
PrintKeysAndValues1( mySSC );

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}

}

/*
This code produces the following output.

```

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

```
System.ArgumentException: value must be no more than 5 characters in length.  
Parameter name: value  
   at ShortStringDictionary.OnValidate(Object key, Object value)  
   at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)  
   at SamplesDictionaryBase.Main()  
  
System.ArgumentException: key must be no more than 5 characters in length.  
Parameter name: key  
   at ShortStringDictionary.OnValidate(Object key, Object value)  
   at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)  
   at SamplesDictionaryBase.Main()  
  
Contains "Three": True  
Contains "Twelve": False  
  
After removing "Two":  
  Three : abc  
  Five  : abcde  
  One   : a  
  Four  : abcd  
  
*/
```

## Remarks

If the [DictionaryBase](#) does not contain an element with the specified key, the [DictionaryBase](#) remains unchanged. No exception is thrown.

This method is an O(1) operation.

See

[Performing Culture-Insensitive String Operations](#)

Also

# DictionaryBase.IDictionary.Values

## In this Article

Gets an [ICollection](#) object containing the values in the [DictionaryBase](#) object.

```
System.Collections.ICollection System.Collections.IDictionary.Values { get; }
```

## Returns

[ICollection](#)

An [ICollection](#) object containing the values in the [DictionaryBase](#) object.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) property of 5 characters or less.

```
using System;
using System.Collections;

public class ShortStringDictionary : DictionaryBase {

    public String this[ String key ] {
        get {
            return( (String) Dictionary[key] );
        }
        set {
            Dictionary[key] = value;
        }
    }

    public ICollection Keys {
        get {
            return( Dictionary.Keys );
        }
    }

    public ICollection Values {
        get {
            return( Dictionary.Values );
        }
    }

    public void Add( String key, String value ) {
        Dictionary.Add( key, value );
    }

    public bool Contains( String key ) {
        return( Dictionary.Contains( key ) );
    }

    public void Remove( String key ) {
        Dictionary.Remove( key );
    }

    protected override void OnInsert( Object key, Object value ) {
        if ( key.GetType() != typeof(System.String) )
            throw new ArgumentException( "key must be of type String.", "key" );
        else {
            String strKey = (String) key;
            if ( strKey.Length > 5 )
                throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
        }
    }
}
```

```
        throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
    }
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( newValue.GetType() != typeof(System.String) )
        throw new ArgumentException( "newValue must be of type String.", "newValue" );
    else {
        String strValue = (String) newValue;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "newValue must be no more than 5 characters in length.",
"newValue" );
    }
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key"
);
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
    }
}
}
```

```

public class SamplesDictionaryBase {

    public static void Main() {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );

        // Displays the contents of the collection.
        Console.WriteLine( "After removing \"Two\":" );
        PrintKeysAndValues1( mySSC );

    }

    // Uses the foreach statement which hides the complexity of the enumerator.
    // NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
    public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
        foreach ( DictionaryEntry myDE in myCol )
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    }
}

```

```

        Console.WriteLine();
    }

    // Uses the enumerator.
    // NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
    public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
        DictionaryEntry myDE;
        System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
        while ( myEnumerator.MoveNext() )
            if ( myEnumerator.Current != null ) {
                myDE = (DictionaryEntry) myEnumerator.Current;
                Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
            }
        Console.WriteLine();
    }

    // Uses the Keys property and the Item property.
    public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
        ICollection myKeys = myCol.Keys;
        foreach ( String k in myKeys )
            Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
        Console.WriteLine();
    }

}

/*
This code produces the following output.

```

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

`System.ArgumentException: value must be no more than 5 characters in length.`

Parameter name: value

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

`System.ArgumentException: key must be no more than 5 characters in length.`

Parameter name: key

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

Contains "Three": True

Contains "Twelve": False

After removing "Two":

```
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/
```

## Remarks

The order of the values in the [ICollection](#) object is unspecified, but is the same order as the associated keys in the [ICollection](#) object returned by the [Keys](#) property.

The returned [ICollection](#) is not a static copy; instead, the [ICollection](#) refers back to the values in the original [DictionaryBase](#) object. Therefore, changes to the [DictionaryBase](#) continue to be reflected in the returned [ICollection](#).

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

See

[ICollection](#)

Also

# DictionaryBase.IEnumerable.GetEnumerator

## In this Article

Returns an [IEnumerator](#) that iterates through the [DictionaryBase](#).

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

Returns

[IEnumerator](#)

An [IEnumerator](#) for the [DictionaryBase](#).

## 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, calling [Current](#) throws an exception. 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`, calling [Current](#) throws an exception. 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](#). If the collection is modified between [MoveNext](#) and [Current](#), [Current](#) returns the element that it is set to, even if the enumerator is already invalidated.

The enumerator does not have exclusive access to the collection; therefore, 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 causes 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.

This method is an O(1) operation.

See

[IDictionaryEnumerator](#)

Also

[IEnumerator](#)

# DictionaryBase.InnerHashtable DictionaryBase.InnerHashtable

## In this Article

Gets the list of elements contained in the [DictionaryBase](#) instance.

```
protected System.Collections.Hashtable InnerHashtable { get; }  
member this.InnerHashtable : System.Collections.Hashtable
```

Returns

[Hashtable](#) [Hashtable](#)

A [Hashtable](#) representing the [DictionaryBase](#) instance itself.

## Remarks

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

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

# DictionaryBase.OnClear DictionaryBase.OnClear

## In this Article

Performs additional custom processes before clearing the contents of the [DictionaryBase](#) instance.

```
protected virtual void OnClear ();  
  
abstract member OnClear : unit -> unit  
override this.OnClear : unit -> unit
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the collection is cleared.

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See

[OnClearComplete\(\)](#)[OnClearComplete\(\)](#)

Also

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

# DictionaryBase.OnClearComplete DictionaryBase.OnClearComplete

## In this Article

Performs additional custom processes after clearing the contents of the [DictionaryBase](#) instance.

```
protected virtual void OnClearComplete ();  
  
abstract member OnClearComplete : unit -> unit  
override this.OnClearComplete : unit -> unit
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the collection is cleared.

The `On*` methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See

[OnClear\(\)OnClear\(\)](#)

Also

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

# DictionaryBase.OnGet DictionaryBase.OnGet

## In this Article

Gets the element with the specified key and value in the [DictionaryBase](#) instance.

```
protected virtual object OnGet (object key, object currentValue);  
  
abstract member OnGet : obj * obj -> obj  
override this.OnGet : obj * obj -> obj
```

## Parameters

**key** [Object Object](#)

The key of the element to get.

**currentValue** [Object Object](#)

The current value of the element associated with **key**.

## Returns

[Object Object](#)

An [Object](#) containing the element with the specified key and value.

## Remarks

The default implementation of this method returns **currentValue**. It is intended to be overridden by a derived class to perform additional action when the specified element is retrieved.

The **On\*** methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

# DictionaryBase.OnInsert DictionaryBase.OnInsert

## In this Article

Performs additional custom processes before inserting a new element into the [DictionaryBase](#) instance.

```
protected virtual void OnInsert (object key, object value);  
  
abstract member OnInsert : obj * obj -> unit  
override this.OnInsert : obj * obj -> unit
```

## Parameters

key	<a href="#">Object</a>
The key of the element to insert.	
value	<a href="#">Object</a>
The value of the element to insert.	

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;  
using System.Collections;  
  
public class ShortStringDictionary : DictionaryBase {  
  
    public String this[ String key ] {  
        get {  
            return( (String) Dictionary[key] );  
        }  
        set {  
            Dictionary[key] = value;  
        }  
    }  
  
    public ICollection Keys {  
        get {  
            return( Dictionary.Keys );  
        }  
    }  
  
    public ICollection Values {  
        get {  
            return( Dictionary.Values );  
        }  
    }  
  
    public void Add( String key, String value ) {  
        Dictionary.Add( key, value );  
    }  
  
    public bool Contains( String key ) {  
        return( Dictionary.Contains( key ) );  
    }  
  
    public void Remove( String key ) {  
        Dictionary.Remove( key );  
    }  
}
```

```

protected override void OnInsert( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
    }
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( newValue.GetType() != typeof(System.String) )
        throw new ArgumentException( "newValue must be of type String.", "newValue" );
    else {
        String strValue = (String) newValue;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
    }
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {

```

```

        stringValue = (String) value;
        if ( stringValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.", 
"value" );
    }
}

public class SamplesDictionaryBase  {

    public static void Main()  {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try  {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try  {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );

        // Displays the contents of the collection.
        Console.WriteLine( "After removing \"Two\":" );
        PrintKeysAndValues1( mySSC );
    }
}

```

```

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

System.ArgumentException: value must be no more than 5 characters in length.

Parameter name: value

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

System.ArgumentException: key must be no more than 5 characters in length.

Parameter name: key

```
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

Contains "Three": True
Contains "Twelve": False

After removing "Two":
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the specified element is inserted.

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See

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

Also

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

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

# DictionaryBase.OnInsertComplete DictionaryBase.OnInsertComplete

## In this Article

Performs additional custom processes after inserting a new element into the [DictionaryBase](#) instance.

```
protected virtual void OnInsertComplete (object key, object value);  
  
abstract member OnInsertComplete : obj * obj -> unit  
override this.OnInsertComplete : obj * obj -> unit
```

## Parameters

key [Object](#) [Object](#)

The key of the element to insert.

value [Object](#) [Object](#)

The value of the element to insert.

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the specified element is inserted.

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See [OnInsert\(Object, Object\)](#) [OnInsert\(Object, Object\)](#)

Also [OnSetComplete\(Object, Object, Object\)](#) [OnSetComplete\(Object, Object, Object\)](#)

# DictionaryBase.OnRemove DictionaryBase.OnRemove

## In this Article

Performs additional custom processes before removing an element from the [DictionaryBase](#) instance.

```
protected virtual void OnRemove (object key, object value);  
  
abstract member OnRemove : obj * obj -> unit  
override this.OnRemove : obj * obj -> unit
```

## Parameters

key [Object](#) [Object](#)

The key of the element to remove.

value [Object](#) [Object](#)

The value of the element to remove.

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;  
using System.Collections;  
  
public class ShortStringDictionary : DictionaryBase {  
  
    public String this[ String key ] {  
        get {  
            return( (String) Dictionary[key] );  
        }  
        set {  
            Dictionary[key] = value;  
        }  
    }  
  
    public ICollection Keys {  
        get {  
            return( Dictionary.Keys );  
        }  
    }  
  
    public ICollection Values {  
        get {  
            return( Dictionary.Values );  
        }  
    }  
  
    public void Add( String key, String value ) {  
        Dictionary.Add( key, value );  
    }  
  
    public bool Contains( String key ) {  
        return( Dictionary.Contains( key ) );  
    }  
  
    public void Remove( String key ) {  
        Dictionary.Remove( key );  
    }  
}
```

```

protected override void OnInsert( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
    }
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( newValue.GetType() != typeof(System.String) )
        throw new ArgumentException( "newValue must be of type String.", "newValue" );
    else {
        String strValue = (String) newValue;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
    }
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {

```

```

        stringValue = (String) value;
        if ( stringValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.", 
"value" );
    }
}

public class SamplesDictionaryBase  {

    public static void Main()  {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try  {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try  {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );

        // Displays the contents of the collection.
        Console.WriteLine( "After removing \"Two\":" );
        PrintKeysAndValues1( mySSC );
    }
}

```

```

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

System.ArgumentException: value must be no more than 5 characters in length.

Parameter name: value

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

System.ArgumentException: key must be no more than 5 characters in length.

Parameter name: key

```
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

Contains "Three": True
Contains "Twelve": False

After removing "Two":
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the specified element is removed.

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See

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

Also

[OnClear\(\)](#)

# DictionaryBase.OnRemoveComplete DictionaryBase.OnRemoveComplete

## In this Article

Performs additional custom processes after removing an element from the [DictionaryBase](#) instance.

```
protected virtual void OnRemoveComplete (object key, object value);  
  
abstract member OnRemoveComplete : obj * obj -> unit  
override this.OnRemoveComplete : obj * obj -> unit
```

## Parameters

key [Object](#) [Object](#)

The key of the element to remove.

value [Object](#) [Object](#)

The value of the element to remove.

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the specified element is removed.

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See [OnRemove\(Object, Object\)](#) [OnRemove\(Object, Object\)](#)

Also [OnClearComplete\(\)](#) [OnClearComplete\(\)](#)

# DictionaryBase.OnSet DictionaryBase.OnSet

## In this Article

Performs additional custom processes before setting a value in the [DictionaryBase](#) instance.

```
protected virtual void OnSet (object key, object oldValue, object newValue);  
  
abstract member OnSet : obj * obj * obj -> unit  
override this.OnSet : obj * obj * obj -> unit
```

## Parameters

key	Object Object
The key of the element to locate.	
oldValue	Object Object
The old value of the element associated with <code>key</code> .	
newValue	Object Object
The new value of the element associated with <code>key</code> .	

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;  
using System.Collections;  
  
public class ShortStringDictionary : DictionaryBase {  
  
    public String this[ String key ] {  
        get {  
            return( (String) Dictionary[key] );  
        }  
        set {  
            Dictionary[key] = value;  
        }  
    }  
  
    public ICollection Keys {  
        get {  
            return( Dictionary.Keys );  
        }  
    }  
  
    public ICollection Values {  
        get {  
            return( Dictionary.Values );  
        }  
    }  
  
    public void Add( String key, String value ) {  
        Dictionary.Add( key, value );  
    }  
  
    public bool Contains( String key ) {  
        return( Dictionary.Contains( key ) );  
    }  
}
```

```

public void Remove( String key ) {
    Dictionary.Remove( key );
}

protected override void OnInsert( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( value.GetType() != typeof(System.String) )
    throw new ArgumentException( "value must be of type String.", "value" );
else {
    String strValue = (String) value;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
}
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

if ( newValue.GetType() != typeof(System.String) )
    throw new ArgumentException( "newValue must be of type String.", "newValue" );
else {
    String strValue = (String) newValue;
    if ( strValue.Length > 5 )
        throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
}
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}
}

```

```

        if ( value.GetType() != typeof(System.String) )
            throw new ArgumentException( "value must be of type String.", "value" );
        else {
            String strValue = (String) value;
            if ( strValue.Length > 5 )
                throw new ArgumentException( "value must be no more than 5 characters in length.",
"value" );
        }
    }

}

public class SamplesDictionaryBase {

    public static void Main() {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e ) {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );
    }
}

```

```

// Displays the contents of the collection.
Console.WriteLine( "After removing \"Two\":");
PrintKeysAndValues1( mySSC );

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

System.ArgumentException: value must be no more than 5 characters in length.

Parameter name: value  
at ShortStringDictionary.OnValidate(Object key, Object value)

```
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()
System.ArgumentException: key must be no more than 5 characters in length.
Parameter name: key
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

Contains "Three": True
Contains "Twelve": False

After removing "Two":
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action before the specified element is set.

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See	<a href="#">OnSetComplete(Object, Object, Object)</a> <a href="#">OnSetComplete(Object, Object, Object)</a>
Also	<a href="#">OnInsert(Object, Object)</a> <a href="#">OnInsert(Object, Object)</a> <a href="#">OnValidate(Object, Object)</a> <a href="#">OnValidate(Object, Object)</a>

# DictionaryBase.OnSetComplete DictionaryBase.OnSetComplete

## In this Article

Performs additional custom processes after setting a value in the [DictionaryBase](#) instance.

```
protected virtual void OnSetComplete (object key, object oldValue, object newValue);  
abstract member OnSetComplete : obj * obj * obj -> unit  
override this.OnSetComplete : obj * obj * obj -> unit
```

## Parameters

key	<a href="#">Object</a>
The key of the element to locate.	
oldValue	<a href="#">Object</a>
The old value of the element associated with <code>key</code> .	
newValue	<a href="#">Object</a>
The new value of the element associated with <code>key</code> .	

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action after the specified element is set.

The `On*` methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See	<a href="#">OnSet(Object, Object, Object)</a>
Also	<a href="#">OnInsertComplete(Object, Object)</a>

# DictionaryBase.OnValidate DictionaryBase.OnValidate

## In this Article

Performs additional custom processes when validating the element with the specified key and value.

```
protected virtual void OnValidate (object key, object value);  
  
abstract member OnValidate : obj * obj -> unit  
override this.OnValidate : obj * obj -> unit
```

## Parameters

key	Object Object
The key of the element to validate.	
value	Object Object
The value of the element to validate.	

## Examples

The following code example implements the [DictionaryBase](#) class and uses that implementation to create a dictionary of [String](#) keys and values that have a [Length](#) of 5 characters or less.

```
using System;  
using System.Collections;  
  
public class ShortStringDictionary : DictionaryBase {  
  
    public String this[ String key ] {  
        get {  
            return( (String) Dictionary[key] );  
        }  
        set {  
            Dictionary[key] = value;  
        }  
    }  
  
    public ICollection Keys {  
        get {  
            return( Dictionary.Keys );  
        }  
    }  
  
    public ICollection Values {  
        get {  
            return( Dictionary.Values );  
        }  
    }  
  
    public void Add( String key, String value ) {  
        Dictionary.Add( key, value );  
    }  
  
    public bool Contains( String key ) {  
        return( Dictionary.Contains( key ) );  
    }  
  
    public void Remove( String key ) {  
        Dictionary.Remove( key );  
    }  
}
```

```

protected override void OnInsert( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {
        String strValue = (String) value;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.", "value" );
    }
}

protected override void OnRemove( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }
}

protected override void OnSet( Object key, Object oldValue, Object newValue ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( newValue.GetType() != typeof(System.String) )
        throw new ArgumentException( "newValue must be of type String.", "newValue" );
    else {
        String strValue = (String) newValue;
        if ( strValue.Length > 5 )
            throw new ArgumentException( "newValue must be no more than 5 characters in length.", "newValue" );
    }
}

protected override void OnValidate( Object key, Object value ) {
    if ( key.GetType() != typeof(System.String) )
        throw new ArgumentException( "key must be of type String.", "key" );
    else {
        String strKey = (String) key;
        if ( strKey.Length > 5 )
            throw new ArgumentException( "key must be no more than 5 characters in length.", "key" );
    }

    if ( value.GetType() != typeof(System.String) )
        throw new ArgumentException( "value must be of type String.", "value" );
    else {

```

```

        stringValue = (String) value;
        if ( stringValue.Length > 5 )
            throw new ArgumentException( "value must be no more than 5 characters in length.", 
"value" );
    }
}

public class SamplesDictionaryBase  {

    public static void Main()  {

        // Creates and initializes a new DictionaryBase.
        ShortStringDictionary mySSC = new ShortStringDictionary();

        // Adds elements to the collection.
        mySSC.Add( "One", "a" );
        mySSC.Add( "Two", "ab" );
        mySSC.Add( "Three", "abc" );
        mySSC.Add( "Four", "abcd" );
        mySSC.Add( "Five", "abcde" );

        // Display the contents of the collection using foreach. This is the preferred method.
        Console.WriteLine( "Contents of the collection (using foreach):" );
        PrintKeysAndValues1( mySSC );

        // Display the contents of the collection using the enumerator.
        Console.WriteLine( "Contents of the collection (using enumerator):" );
        PrintKeysAndValues2( mySSC );

        // Display the contents of the collection using the Keys property and the Item property.
        Console.WriteLine( "Initial contents of the collection (using Keys and Item):" );
        PrintKeysAndValues3( mySSC );

        // Tries to add a value that is too long.
        try  {
            mySSC.Add( "Ten", "abcdefghijkl" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        // Tries to add a key that is too long.
        try  {
            mySSC.Add( "Eleven", "ijk" );
        }
        catch ( ArgumentException e )  {
            Console.WriteLine( e.ToString() );
        }

        Console.WriteLine();

        // Searches the collection with Contains.
        Console.WriteLine( "Contains \"Three\": {0}", mySSC.Contains( "Three" ) );
        Console.WriteLine( "Contains \"Twelve\": {0}", mySSC.Contains( "Twelve" ) );
        Console.WriteLine();

        // Removes an element from the collection.
        mySSC.Remove( "Two" );

        // Displays the contents of the collection.
        Console.WriteLine( "After removing \"Two\":" );
        PrintKeysAndValues1( mySSC );
    }
}

```

```

}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues1( ShortStringDictionary myCol ) {
    foreach ( DictionaryEntry myDE in myCol )
        Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintKeysAndValues2( ShortStringDictionary myCol ) {
    DictionaryEntry myDE;
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        if ( myEnumerator.Current != null ) {
            myDE = (DictionaryEntry) myEnumerator.Current;
            Console.WriteLine( " {0,-5} : {1}", myDE.Key, myDE.Value );
        }
    Console.WriteLine();
}

// Uses the Keys property and the Item property.
public static void PrintKeysAndValues3( ShortStringDictionary myCol ) {
    ICollection myKeys = myCol.Keys;
    foreach ( String k in myKeys )
        Console.WriteLine( " {0,-5} : {1}", k, myCol[k] );
    Console.WriteLine();
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Contents of the collection (using enumerator):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

Initial contents of the collection (using Keys and Item):

```

Three : abc
Five  : abcde
Two   : ab
One   : a
Four  : abcd

```

System.ArgumentException: value must be no more than 5 characters in length.

Parameter name: value

```

at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

```

System.ArgumentException: key must be no more than 5 characters in length.

Parameter name: key

```
at ShortStringDictionary.OnValidate(Object key, Object value)
at System.Collections.DictionaryBase.System.Collections.IDictionary.Add(Object key, Object value)
at SamplesDictionaryBase.Main()

Contains "Three": True
Contains "Twelve": False

After removing "Two":
Three : abc
Five  : abcde
One   : a
Four  : abcd

*/
```

## Remarks

The default implementation of this method is intended to be overridden by a derived class to perform some action when the specified element is validated.

The On\* methods are invoked only on the instance returned by the [Dictionary](#) property, but not on the instance returned by the [InnerHashtable](#) property.

The default implementation of this method is an O(1) operation.

See

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

Also

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

# DictionaryEntry DictionaryEntry Struct

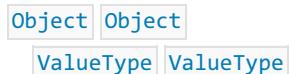
Defines a dictionary key/value pair that can be set or retrieved.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public struct DictionaryEntry

type DictionaryEntry = struct
```

## Inheritance Hierarchy



## Remarks

The [IDictionaryEnumerator.Entry](#) method returns an instance of this type.

### Important

We don't recommend that you use the `DictionaryEntry` structure for new development. Instead, we recommend that you use a generic `KeyValuePair<TKey,TValue>` structure along with the `Dictionary<TKey,TValue>` class. For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

The C# `foreach` statement and the Visual Basic `For Each` statement require the type of each element in the collection. Since each element of the `IDictionary` is a key/value pair, the element type is not the type of the key or the type of the value. Instead, the element type is `DictionaryEntry`. For example:

```
foreach (DictionaryEntry de in openWith)
{
    Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
}
```

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

## Constructors

`DictionaryEntry(Object, Object)`

`DictionaryEntry(Object, Object)`

Initializes an instance of the `DictionaryEntry` type with the specified key and value.

## Properties

`Key`

`Key`

Gets or sets the key in the key/value pair.

`Value`

## Value

---

Gets or sets the value in the key/value pair.

## Methods

Deconstruct(Object, Object)

Deconstruct(Object, Object)

---

# DictionaryEntry.Deconstruct DictionaryEntry.Deconstruct

## In this Article

```
public void Deconstruct (out object key, out object value);  
member this.Deconstruct : * -> unit
```

## Parameters

key	Object	Object
value	Object	Object

# DictionaryEntry DictionaryEntry

## In this Article

Initializes an instance of the [DictionaryEntry](#) type with the specified key and value.

```
public DictionaryEntry (object key, object value);  
new System.Collections.DictionaryEntry : obj * obj -> System.Collections.DictionaryEntry
```

### Parameters

key Object Object

The object defined in each key/value pair.

value Object Object

The definition associated with `key`.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null` and the .NET Framework version is 1.0 or 1.1.

## Remarks

In the .NET Framework 2.0 `key` can be `null`.

# DictionaryEntry.Key DictionaryEntry.Key

## In this Article

Gets or sets the key in the key/value pair.

```
public object Key { get; set; }  
member this.Key : obj with get, set
```

Returns

[Object Object](#)

The key in the key/value pair.

## Examples

The following example demonstrates the [Key](#) property. This code example is part of a larger example provided for the [DictionaryEntry](#) class.

```
public void Add(object key, object value)  
{  
    // Add the new key/value pair even if this key already exists in the dictionary.  
    if (ItemsInUse == items.Length)  
        throw new InvalidOperationException("The dictionary cannot hold any more items.");  
    items[ItemsInUse++] = new DictionaryEntry(key, value);  
}
```

# DictionaryEntry.Value DictionaryEntry.Value

## In this Article

Gets or sets the value in the key/value pair.

```
public object Value { get; set; }  
member this.Value : obj with get, set
```

Returns

[Object Object](#)

The value in the key/value pair.

## Examples

The following example demonstrates the [Value](#) property. This code example is part of a larger example provided for the [DictionaryEntry](#) class.

```
public void Add(object key, object value)  
{  
    // Add the new key/value pair even if this key already exists in the dictionary.  
    if (ItemsInUse == items.Length)  
        throw new InvalidOperationException("The dictionary cannot hold any more items.");  
    items[ItemsInUse++] = new DictionaryEntry(key, value);  
}
```

# Hashtable Class

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

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public class Hashtable : ICloneable, System.Collections.IDictionary,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable

type Hashtable = class
    interface IDictionary
    interface ISerializable
    interface IDeserializationCallback
    interface ICloneable
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

Each element is a key/value pair stored in a [DictionaryEntry](#) object. A key cannot be `null`, but a value can be.

**Important**

We don't recommend that you use the [Hashtable](#) class for new development. Instead, we recommend that you use the generic [HashSet<T>](#) class. For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

The objects used as keys by a [Hashtable](#) are required to override the [Object.GetHashCode](#) method (or the [IHashCodeProvider](#) interface) and the [Object.Equals](#) method (or the [IComparer](#) interface). The implementation of both methods and interfaces must handle case sensitivity the same way; otherwise, the [Hashtable](#) might behave incorrectly. For example, when creating a [Hashtable](#), you must use the [CaseInsensitiveHashCodeProvider](#) class (or any case-insensitive [IHashCodeProvider](#) implementation) with the [CaseInsensitiveComparer](#) class (or any case-insensitive [IComparer](#) implementation).

Furthermore, these methods must produce the same results when called with the same parameters while the key exists in the [Hashtable](#). An alternative is to use a [Hashtable](#) constructor with an [IEqualityComparer](#) parameter. If key equality were simply reference equality, the inherited implementation of [Object.GetHashCode](#) and [Object.Equals](#) would suffice.

Key objects must be immutable as long as they are used as keys in the [Hashtable](#).

When an element is added to the [Hashtable](#), the element is placed into a bucket based on the hash code of the key. Subsequent lookups of the key use the hash code of the key to search in only one particular bucket, thus substantially reducing the number of key comparisons required to find an element.

The load factor of a [Hashtable](#) determines the maximum ratio of elements to buckets. Smaller load factors cause faster average lookup times at the cost of increased memory consumption. The default load factor of 1.0 generally provides the best balance between speed and size. A different load factor can also be specified when the [Hashtable](#) is created.

As elements are added to a [Hashtable](#), the actual load factor of the [Hashtable](#) increases. When the actual load factor reaches the specified load factor, the number of buckets in the [Hashtable](#) is automatically increased to the smallest prime number that is larger than twice the current number of [Hashtable](#) buckets.

Each key object in the [Hashtable](#) must provide its own hash function, which can be accessed by calling [GetHash](#).

However, any object implementing [IHashCodeProvider](#) can be passed to a [Hashtable](#) constructor, and that hash function is used for all objects in the table.

The capacity of a [Hashtable](#) is the number of elements the [Hashtable](#) can hold. As elements are added to a [Hashtable](#), the capacity is automatically increased as required through reallocation.

For very large [Hashtable](#) objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the `enabled` attribute of the configuration element to `true` in the run-time environment.

The `foreach` statement of the C# language (`For Each` in Visual Basic) returns an object of the type of the elements in the collection. Since each element of the [Hashtable](#) is a key/value pair, the element type is not the type of the key or the type of the value. Instead, the element type is [DictionaryEntry](#). For example:

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

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

Because serializing and deserializing an enumerator for a [Hashtable](#) can cause the elements to become reordered, it is not possible to continue enumeration without calling the [Reset](#) method.

**Note**

Because keys can be inherited and their behavior changed, their absolute uniqueness cannot be guaranteed by comparisons using the [Equals](#) method.

## Constructors

[Hashtable\(\)](#)

[Hashtable\(\)](#)

Initializes a new, empty instance of the [Hashtable](#) class using the default initial capacity, load factor, hash code provider, and comparer.

[Hashtable\(Int32, Single, IHashCodeProvider, IComparer\)](#)

[Hashtable\(Int32, Single, IHashCodeProvider, IComparer\)](#)

Initializes a new, empty instance of the [Hashtable](#) class using the specified initial capacity, load factor, hash code provider, and comparer.

[Hashtable\(IDictionary, Single, IHashCodeProvider, IComparer\)](#)

[Hashtable\(IDictionary, Single, IHashCodeProvider, IComparer\)](#)

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the specified load factor, hash code provider, and comparer.

[Hashtable\(Int32, Single, IEqualityComparer\)](#)

[Hashtable\(Int32, Single, IEqualityComparer\)](#)

Initializes a new, empty instance of the [Hashtable](#) class using the specified initial capacity, load factor, and

[IEqualityComparer](#) object.

```
Hashtable(Int32, IHashCodeProvider, IComparer)  
Hashtable(Int32, IHashCodeProvider, IComparer)
```

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

```
Hashtable(IDictionary, Single, IEqualityComparer)  
Hashtable(IDictionary, Single, IEqualityComparer)
```

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the specified load factor and [IEqualityComparer](#) object.

```
Hashtable(IDictionary, IHashCodeProvider, IComparer)  
Hashtable(IDictionary, IHashCodeProvider, IComparer)
```

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the default load factor, and the specified hash code provider and comparer. This API is obsolete. For an alternative, see [Hashtable\(IDictionary, IEqualityComparer\)](#).

```
Hashtable(Int32, Single)  
Hashtable(Int32, Single)
```

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

```
Hashtable(SerializationInfo, StreamingContext)  
Hashtable(SerializationInfo, StreamingContext)
```

Initializes a new, empty instance of the [Hashtable](#) class that is serializable using the specified [SerializationInfo](#) and [StreamingContext](#) objects.

```
Hashtable(IHashCodeProvider, IComparer)  
Hashtable(IHashCodeProvider, IComparer)
```

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

```
Hashtable(IDictionary, Single)  
Hashtable(IDictionary, Single)
```

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and

uses the specified load factor, and the default hash code provider and comparer.

```
Hashtable(IDictionary, IEqualityComparer)  
Hashtable(IDictionary, IEqualityComparer)
```

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to a new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the default load factor and the specified [IEqualityComparer](#) object.

```
Hashtable(Int32)  
Hashtable(Int32)
```

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

```
Hashtable(IEqualityComparer)  
Hashtable(IEqualityComparer)
```

Initializes a new, empty instance of the [Hashtable](#) class using the default initial capacity and load factor, and the specified [IEqualityComparer](#) object.

```
Hashtable(IDictionary)  
Hashtable(IDictionary)
```

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the default load factor, hash code provider, and comparer.

```
Hashtable(Int32, IEqualityComparer)  
Hashtable(Int32, IEqualityComparer)
```

Initializes a new, empty instance of the [Hashtable](#) class using the specified initial capacity and [IEqualityComparer](#), and the default load factor.

## Properties

comparer  
comparer

Gets or sets the [IComparer](#) to use for the [Hashtable](#).

Count  
Count

Gets the number of key/value pairs contained in the [Hashtable](#).

**EqualityComparer**

**EqualityComparer**

Gets the [IEqualityComparer](#) to use for the [Hashtable](#).

**hcp**

**hcp**

Gets or sets the object that can dispense hash codes.

**IsFixedSize**

**IsFixedSize**

Gets a value indicating whether the [Hashtable](#) has a fixed size.

**IsReadOnly**

**IsReadOnly**

Gets a value indicating whether the [Hashtable](#) is read-only.

**IsSynchronized**

**IsSynchronized**

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

**Item[Object]**

**Item[Object]**

Gets or sets the value associated with the specified key.

**Keys**

**Keys**

Gets an [ICollection](#) containing the keys in the [Hashtable](#).

**SyncRoot**

**SyncRoot**

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

**Values**

**Values**

Gets an [ICollection](#) containing the values in the [Hashtable](#).

## Methods

Add(Object, Object)

Add(Object, Object)

Adds an element with the specified key and value into the [Hashtable](#).

Clear()

Clear()

Removes all elements from the [Hashtable](#).

Clone()

Clone()

Creates a shallow copy of the [Hashtable](#).

Contains(Object)

Contains(Object)

Determines whether the [Hashtable](#) contains a specific key.

ContainsKey(Object)

ContainsKey(Object)

Determines whether the [Hashtable](#) contains a specific key.

ContainsValue(Object)

ContainsValue(Object)

Determines whether the [Hashtable](#) contains a specific value.

CopyTo(Array, Int32)

CopyTo(Array, Int32)

Copies the [Hashtable](#) elements to a one-dimensional [Array](#) instance at the specified index.

GetEnumerator()

GetEnumerator()

Returns an [IDictionaryEnumerator](#) that iterates through the [Hashtable](#).

GetHash(Object)

GetHash(Object)

Returns the hash code for the specified key.

```
GetObjectData(SerializationInfo, StreamingContext)
```

```
GetObjectData(SerializationInfo, StreamingContext)
```

Implements the [ISerializable](#) interface and returns the data needed to serialize the [Hashtable](#).

```
KeyEquals(Object, Object)
```

```
KeyEquals(Object, Object)
```

Compares a specific [Object](#) with a specific key in the [Hashtable](#).

```
OnDeserialization(Object)
```

```
OnDeserialization(Object)
```

Implements the [ISerializable](#) interface and raises the deserialization event when the deserialization is complete.

```
Remove(Object)
```

```
Remove(Object)
```

Removes the element with the specified key from the [Hashtable](#).

```
Synchronized(Hashtable)
```

```
Synchronized(Hashtable)
```

Returns a synchronized (thread-safe) wrapper for the [Hashtable](#).

```
IEnumerable.GetEnumerator()
```

```
IEnumerable.GetEnumerator()
```

Returns an enumerator that iterates through a collection.

## Thread Safety

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

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 causes 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.

## See Also

[IDictionary](#) [IDictionary](#)

[IHashCodeProvider](#) [IHashCodeProvider](#)

`GetHashCode() GetHashCode()`  
`Equals(Object) Equals(Object)`  
`IEqualityComparer IEqualityComparer`

# Hashtable.Add Hashtable.Add

## In this Article

Adds an element with the specified key and value into the [Hashtable](#).

```
public virtual void Add (object key, object value);  
  
abstract member Add : obj * obj -> unit  
override this.Add : obj * obj -> unit
```

## Parameters

key	Object Object
-----	---------------

The key of the element to add.

value	Object Object
-------	---------------

The value of the element to add. The value can be `null`.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`key` is `null`.

[ArgumentException](#) [ArgumentException](#)

An element with the same key already exists in the [Hashtable](#).

[NotSupportedException](#) [NotSupportedException](#)

The [Hashtable](#) is read-only.

-or-

The [Hashtable](#) has a fixed size.

## Examples

The following example shows how to add elements to the [Hashtable](#).

```

using System;
using System.Collections;
public class SamplesHashtable {
    public static void Main() {
        // Creates and initializes a new Hashtable.
        Hashtable myHT = new Hashtable();
        myHT.Add( "one", "The" );
        myHT.Add( "two", "quick" );
        myHT.Add( "three", "brown" );
        myHT.Add( "four", "fox" );

        // Displays the Hashtable.
        Console.WriteLine( "The Hashtable contains the following:" );
        PrintKeysAndValues( myHT );
    }

    public static void PrintKeysAndValues( Hashtable myHT ) {
        Console.WriteLine( "      -KEY-      -VALUE-" );
        foreach ( DictionaryEntry de in myHT )
            Console.WriteLine( "      {0}:      {1}", de.Key, de.Value );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The Hashtable contains the following:
      -KEY-      -VALUE-
      two:      quick
      three:    brown
      four:     fox
      one:      The
*/

```

## Remarks

A key cannot be `null`, but a value can be.

An object that has no correlation between its state and its hash code value should typically not be used as the key. For example, `String` objects are better than `StringBuilder` objects for use as keys.

You can also use the `Item[Object]` property to add new elements by setting the value of a key that does not exist in the `Hashtable`; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the `Hashtable`, setting the `Item[Object]` property overwrites the old value. In contrast, the `Add` method does not modify existing elements.

If `Count` is less than the capacity of the `Hashtable`, 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`.

See

[Remove\(Object\)](#)

Also

[Item\[Object\]](#)

[Item\[Object\]](#)

# Hashtable.Clear Hashtable.Clear

## In this Article

Removes all elements from the [Hashtable](#).

```
public virtual void Clear ();  
  
abstract member Clear : unit -> unit  
override this.Clear : unit -> unit
```

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [Hashtable](#) is read-only.

## Examples

The following example shows how to clear the values of the [Hashtable](#).

```

using System;
using System.Collections;
public class SamplesHashtable {
    public static void Main() {
        // Creates and initializes a new Hashtable.
        Hashtable myHT = new Hashtable();
        myHT.Add( "one", "The" );
        myHT.Add( "two", "quick" );
        myHT.Add( "three", "brown" );
        myHT.Add( "four", "fox" );
        myHT.Add( "five", "jumps" );

        // Displays the count and values of the Hashtable.
        Console.WriteLine( "Initially," );
        Console.WriteLine( "    Count      : {0}", myHT.Count );
        Console.WriteLine( "    Values:" );
        PrintKeysAndValues( myHT );

        // Clears the Hashtable.
        myHT.Clear();

        // Displays the count and values of the Hashtable.
        Console.WriteLine( "After Clear," );
        Console.WriteLine( "    Count      : {0}", myHT.Count );
        Console.WriteLine( "    Values:" );
        PrintKeysAndValues( myHT );
    }

    public static void PrintKeysAndValues( Hashtable myHT ) {
        Console.WriteLine( "    -KEY-      -VALUE-" );
        foreach ( DictionaryEntry de in myHT )
            Console.WriteLine( "    {0}:      {1}", de.Key, de.Value );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Initially,
Count      : 5
Values:
    -KEY-      -VALUE-
    two:      quick
    three:    brown
    four:     fox
    five:     jumps
    one:      The

After Clear,
Count      : 0
Values:
    -KEY-      -VALUE-
*/

```

## Remarks

`Count` is set to zero, 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 `Count`.

# Hashtable.Clone Hashtable.Clone

## In this Article

Creates a shallow copy of the [Hashtable](#).

```
public virtual object Clone ();  
  
abstract member Clone : unit -> obj  
override this.Clone : unit -> obj
```

Returns

[Object Object](#)

A shallow copy of the [Hashtable](#).

## Remarks

A shallow copy of a collection copies only the elements of the collection, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new collection point to the same objects that the references in the original collection point to.

In contrast, a deep copy of a collection copies the elements and everything directly or indirectly referenced by the elements.

The [Hashtable](#) clone has the same count, the same capacity, the same [IHashCodeProvider](#) implementation, and the same [IComparer](#) implementation as the original [Hashtable](#).

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

See

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

Also

# Hashtable.comparer Hashtable.comparer

## In this Article

Gets or sets the [IComparer](#) to use for the [Hashtable](#).

```
[System.Obsolete("Please use KeyComparer properties.")]
protected System.Collections.IComparer comparer { get; set; }

member this.comparer : System.Collections.IComparer with get, set
```

Returns

[IComparer](#) [IComparer](#)

The [IComparer](#) to use for the [Hashtable](#).

Attributes

[ObsoleteAttribute](#)

Exceptions

[ArgumentException](#) [ArgumentException](#)

The property is set to a value, but the hash table was created using an [IEqualityComparer](#).

## Remarks

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

See

[IComparer](#)[IComparer](#)

Also

# Hashtable.Contains Hashtable.Contains

## In this Article

Determines whether the [Hashtable](#) contains a specific key.

```
public virtual bool Contains (object key);  
  
abstract member Contains : obj -> bool  
override this.Contains : obj -> bool
```

## Parameters

**key** [Object](#) [Object](#)

The key to locate in the [Hashtable](#).

## Returns

[Boolean](#) [Boolean](#)

`true` if the [Hashtable](#) contains an element with the specified key; otherwise, `false`.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

## Examples

The following example shows how to determine whether the [Hashtable](#) contains a specific element.

```

using System;
using System.Collections;
public class SamplesHashtable {
    public static void Main() {
        // Creates and initializes a new Hashtable.
        Hashtable myHT = new Hashtable();
        myHT.Add( 0, "zero" );
        myHT.Add( 1, "one" );
        myHT.Add( 2, "two" );
        myHT.Add( 3, "three" );
        myHT.Add( 4, "four" );

        // Displays the values of the Hashtable.
        Console.WriteLine( "The Hashtable contains the following values:" );
        PrintIndexAndKeysAndValues( myHT );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"\{0\}\" is {1}.", myKey, myHT.ContainsKey( myKey ) ? "in the Hashtable" : "NOT in the Hashtable" );
        myKey = 6;
        Console.WriteLine( "The key \"\{0\}\" is {1}.", myKey, myHT.ContainsKey( myKey ) ? "in the Hashtable" : "NOT in the Hashtable" );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"\{0\}\" is {1}.", myValue, myHT.ContainsValue( myValue ) ? "in the Hashtable" : "NOT in the Hashtable" );
        myValue = "nine";
        Console.WriteLine( "The value \"\{0\}\" is {1}.", myValue, myHT.ContainsValue( myValue ) ? "in the Hashtable" : "NOT in the Hashtable" );
    }
}

```

```

public static void PrintIndexAndKeysAndValues( Hashtable myHT ) {
    int i = 0;
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    foreach ( DictionaryEntry de in myHT )
        Console.WriteLine( " [{0}]: {1} {2}", i++, de.Key, de.Value );
    Console.WriteLine();
}

```

```

/*
This code produces the following output.

The Hashtable contains the following values:

```

-INDEX-	-KEY-	-VALUE-
[0]:	4	four
[1]:	3	three
[2]:	2	two
[3]:	1	one
[4]:	0	zero

```

The key "2" is in the Hashtable.
The key "6" is NOT in the Hashtable.
The value "three" is in the Hashtable.
The value "nine" is NOT in the Hashtable.
*/

```

## Remarks

`Contains` implements [IDictionary.Contains](#). It behaves exactly as [ContainsKey](#).

This method is an O(1) operation.

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

[ContainsKey\(Object\)](#)

Also

[IDictionary](#)

[IDictionary](#)

# Hashtable.ContainsKey Hashtable.ContainsKey

## In this Article

Determines whether the [Hashtable](#) contains a specific key.

```
public virtual bool ContainsKey (object key);  
  
abstract member ContainsKey : obj -> bool  
override this.ContainsKey : obj -> bool
```

## Parameters

**key** [Object](#) [Object](#)

The key to locate in the [Hashtable](#).

## Returns

[Boolean](#) [Boolean](#)

`true` if the [Hashtable](#) contains an element with the specified key; otherwise, `false`.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

## Examples

The following example shows how to determine whether the [Hashtable](#) contains a specific element.

```

using System;
using System.Collections;
public class SamplesHashtable {
    public static void Main() {
        // Creates and initializes a new Hashtable.
        Hashtable myHT = new Hashtable();
        myHT.Add( 0, "zero" );
        myHT.Add( 1, "one" );
        myHT.Add( 2, "two" );
        myHT.Add( 3, "three" );
        myHT.Add( 4, "four" );

        // Displays the values of the Hashtable.
        Console.WriteLine( "The Hashtable contains the following values:" );
        PrintIndexAndKeysAndValues( myHT );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"\{0\}\" is {1}.", myKey, myHT.ContainsKey( myKey ) ? "in the Hashtable" : "NOT in the Hashtable" );
        myKey = 6;
        Console.WriteLine( "The key \"\{0\}\" is {1}.", myKey, myHT.ContainsKey( myKey ) ? "in the Hashtable" : "NOT in the Hashtable" );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"\{0\}\" is {1}.", myValue, myHT.ContainsValue( myValue ) ? "in the Hashtable" : "NOT in the Hashtable" );
        myValue = "nine";
        Console.WriteLine( "The value \"\{0\}\" is {1}.", myValue, myHT.ContainsValue( myValue ) ? "in the Hashtable" : "NOT in the Hashtable" );
    }
}

```

```

public static void PrintIndexAndKeysAndValues( Hashtable myHT ) {
    int i = 0;
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    foreach ( DictionaryEntry de in myHT )
        Console.WriteLine( " [{0}]: {1} {2}", i++, de.Key, de.Value );
    Console.WriteLine();
}

```

/\*
This code produces the following output.

The Hashtable contains the following values:

-INDEX-	-KEY-	-VALUE-
[0]:	4	four
[1]:	3	three
[2]:	2	two
[3]:	1	one
[4]:	0	zero

The key "2" is in the Hashtable.

The key "6" is NOT in the Hashtable.

The value "three" is in the Hashtable.

The value "nine" is NOT in the Hashtable.

\*/

## Remarks

This method behaves exactly as [Contains](#).

This method is an O(1) operation.

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

[Contains\(Object\)](#)

Also

[ContainsValue\(Object\)](#)

[Contains\(Object\)](#)

[ContainsValue\(Object\)](#)

# Hashtable.ContainsValue Hashtable.ContainsValue

## In this Article

Determines whether the [Hashtable](#) contains a specific value.

```
public virtual bool ContainsValue (object value);  
  
abstract member ContainsValue : obj -> bool  
override this.ContainsValue : obj -> bool
```

## Parameters

**value** [Object](#) [Object](#)

The value to locate in the [Hashtable](#). The value can be `null`.

## Returns

[Boolean](#) [Boolean](#)

`true` if the [Hashtable](#) contains an element with the specified `value`; otherwise, `false`.

## Examples

The following example shows how to determine whether the [Hashtable](#) contains a specific element.

```

using System;
using System.Collections;
public class SamplesHashtable {
    public static void Main() {
        // Creates and initializes a new Hashtable.
        Hashtable myHT = new Hashtable();
        myHT.Add( 0, "zero" );
        myHT.Add( 1, "one" );
        myHT.Add( 2, "two" );
        myHT.Add( 3, "three" );
        myHT.Add( 4, "four" );

        // Displays the values of the Hashtable.
        Console.WriteLine( "The Hashtable contains the following values:" );
        PrintIndexAndKeysAndValues( myHT );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"\{0\}\" is {1}.", myKey, myHT.ContainsKey( myKey ) ? "in the Hashtable" : "NOT in the Hashtable" );
        myKey = 6;
        Console.WriteLine( "The key \"\{0\}\" is {1}.", myKey, myHT.ContainsKey( myKey ) ? "in the Hashtable" : "NOT in the Hashtable" );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"\{0\}\" is {1}.", myValue, myHT.ContainsValue( myValue ) ? "in the Hashtable" : "NOT in the Hashtable" );
        myValue = "nine";
        Console.WriteLine( "The value \"\{0\}\" is {1}.", myValue, myHT.ContainsValue( myValue ) ? "in the Hashtable" : "NOT in the Hashtable" );
    }
}

```

```

public static void PrintIndexAndKeysAndValues( Hashtable myHT ) {
    int i = 0;
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    foreach ( DictionaryEntry de in myHT )
        Console.WriteLine( " [{0}]: {1} {2}", i++, de.Key, de.Value );
    Console.WriteLine();
}

```

```

/*
This code produces the following output.

The Hashtable contains the following values:

```

-INDEX-	-KEY-	-VALUE-
[0]:	4	four
[1]:	3	three
[2]:	2	two
[3]:	1	one
[4]:	0	zero

```

The key "2" is in the Hashtable.
The key "6" is NOT in the Hashtable.
The value "three" is in the Hashtable.
The value "nine" is NOT in the Hashtable.
*/

```

## Remarks

The values of the elements of the [Hashtable](#) are compared to the specified value using the [Object.Equals](#) method.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[ContainsKey\(Object\)](#)

Also

[ContainsKey\(Object\)](#)

[Equals\(Object\)](#)

# Hashtable.CopyTo Hashtable.CopyTo

## In this Article

Copies the [Hashtable](#) elements to a one-dimensional [Array](#) instance at the specified index.

```
public virtual void CopyTo (Array array, int arrayIndex);  
  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

## Parameters

array [Array](#) [Array](#)

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

arrayIndex [Int32](#) [Int32](#)

The zero-based index in `array` at which copying begins.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`arrayIndex` is less than zero.

[ArgumentException](#) [ArgumentException](#)

`array` is multidimensional.

-or-

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

[InvalidCastException](#) [InvalidCastException](#)

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

## Examples

The following example shows how to copy the list of keys or the list of values in a [Hashtable](#) into a one-dimensional [Array](#).

```

using System;
using System.Collections;
public class SamplesHashtable {

    public static void Main() {

        // Creates and initializes the source Hashtable.
        Hashtable mySourceHT = new Hashtable();
        mySourceHT.Add( "A", "valueA" );
        mySourceHT.Add( "B", "valueB" );

        // Creates and initializes the one-dimensional target Array.
        String[] myTargetArray = new String[15];
        myTargetArray[0] = "The";
        myTargetArray[1] = "quick";
        myTargetArray[2] = "brown";
        myTargetArray[3] = "fox";
        myTargetArray[4] = "jumps";
        myTargetArray[5] = "over";
        myTargetArray[6] = "the";
        myTargetArray[7] = "lazy";
        myTargetArray[8] = "dog";

        // Displays the values of the target Array.
        Console.WriteLine( "The target Array contains the following before:" );
        PrintValues( myTargetArray, ' ' );

        // Copies the keys in the source Hashtable to the target Hashtable, starting at index 6.
        Console.WriteLine( "After copying the keys, starting at index 6:" );
        mySourceHT.Keys.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the values in the source Hashtable to the target Hashtable, starting at index 6.
        Console.WriteLine( "After copying the values, starting at index 6:" );
        mySourceHT.Values.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );
    }

    public static void PrintValues( String[] myArr, char mySeparator ) {
        for ( int i = 0; i < myArr.Length; i++ )
            Console.Write( "{0}{1}", mySeparator, myArr[i] );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The target Array contains the following before:
The quick brown fox jumps over the lazy dog
After copying the keys, starting at index 6:
The quick brown fox jumps over B A dog
After copying the values, starting at index 6:
The quick brown fox jumps over valueB valueA dog
*/

```

## Remarks

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [Hashtable](#).

To copy only the keys in the [Hashtable](#), use `Hashtable.Keys.CopyTo`.

To copy only the values in the [Hashtable](#), use `Hashtable.Values.CopyTo`.

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

See

Also

[Array](#)  
[Array](#)

[GetEnumerator\(\)](#)  
[GetEnumerator\(\)](#)

# Hashtable.Count Hashtable.Count

## In this Article

Gets the number of key/value pairs contained in the [Hashtable](#).

```
public virtual int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of key/value pairs contained in the [Hashtable](#).

## Remarks

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

# Hashtable.EqualityComparer Hashtable.EqualityComparer

## In this Article

Gets the [IEqualityComparer](#) to use for the [Hashtable](#).

```
protected System.Collections.IEqualityComparer EqualityComparer { get; }  
member this.EqualityComparer : System.Collections.IEqualityComparer
```

Returns

[IEqualityComparer](#) [IEqualityComparer](#)

The [IEqualityComparer](#) to use for the [Hashtable](#).

Exceptions

[ArgumentException](#) [ArgumentException](#)

The property is set to a value, but the hash table was created using an [IHashCodeProvider](#) and an [IComparer](#).

## Remarks

The [IEqualityComparer](#) includes both the comparer and the hash code provider. If an [IEqualityComparer](#) is used in the [Hashtable](#) constructor, the objects used as keys in the [Hashtable](#) are not required to override the [Object.GetHashCode](#) and [Object.Equals](#) methods.

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

See

[IEqualityComparer](#)[IEqualityComparer](#)

Also

# Hashtable.GetEnumerator Hashtable.GetEnumerator

## In this Article

Returns an [IDictionaryEnumerator](#) that iterates through the [Hashtable](#).

```
public virtual System.Collections.IDictionaryEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IDictionaryEnumerator  
override this.GetEnumerator : unit -> System.Collections.IDictionaryEnumerator
```

Returns

[IDictionaryEnumerator](#) [IDictionaryEnumerator](#)

An [IDictionaryEnumerator](#) for the [Hashtable](#).

## Examples

The following example compares the use of [GetEnumerator](#) and [foreach](#) to enumerate the contents of a [Hashtable](#).

```
using System;  
using System.Collections;  
  
public class HashtableExample  
{  
    public static void Main()  
    {  
        // Creates and initializes a new Hashtable.  
        Hashtable clouds = new Hashtable();  
        clouds.Add("Cirrus", "Castellanus");  
        clouds.Add("Cirrocumulus", "Stratiformis");  
        clouds.Add("Altocstratus", "Radiatus");  
        clouds.Add("Stratocumulus", "Perlucidus");  
        clouds.Add("Stratus", "Fractus");  
        clouds.Add("Nimbostratus", "Pannus");  
        clouds.Add("Cumulus", "Humilis");  
        clouds.Add("Cumulonimbus", "Incus");  
  
        // Displays the keys and values of the Hashtable using GetEnumerator()  
  
        IDictionaryEnumerator denum = clouds.GetEnumerator();  
        DictionaryEntry dentry;  
  
        Console.WriteLine();  
        Console.WriteLine("    Cloud Type          Variation");  
        Console.WriteLine("    -----");  
        while (denum.MoveNext())  
        {  
            dentry = (DictionaryEntry) denum.Current;  
            Console.WriteLine("    {0,-17}{1}", dentry.Key, dentry.Value);  
        }  
        Console.WriteLine();  
  
        // Displays the keys and values of the Hashtable using foreach statement  
  
        Console.WriteLine("    Cloud Type          Variation");  
        Console.WriteLine("    -----");  
        foreach (DictionaryEntry de in clouds)  
        {  
            Console.WriteLine("    {0,-17}{1}", de.Key, de.Value);  
        }  
        Console.WriteLine();  
    }  
}
```

```

}

// The program displays the following output to the console:
//
//   Cloud Type      Variation
//   -----
//   Cirrocumulus    Stratiformis
//   Stratocumulus   Perlucidus
//   Cirrus          Castellanus
//   Cumulus         Humilis
//   Nimbostratus   Pannus
//   Stratus         Fractus
//   Altostratus    Radiatus
//   Cumulonimbus   Incus

//
//   Cloud Type      Variation
//   -----
//   Cirrocumulus    Stratiformis
//   Stratocumulus   Perlucidus
//   Cirrus          Castellanus
//   Cumulus         Humilis
//   Nimbostratus   Pannus
//   Stratus         Fractus
//   Altostratus    Radiatus
//   Cumulonimbus   Incus*/

```

## Remarks

[Visual Basic, C#]

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 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.

This method is an O(1) operation.

Because serializing and deserializing an enumerator for a `Hashtable` can cause the elements to become reordered, it is not possible to continue enumeration without calling the `Reset` method.

See

[IDictionaryEnumerator](#)[IDictionaryEnumerator](#)

Also

IEnumerator|IEnumerator

# Hashtable.GetHash Hashtable.GetHash

## In this Article

Returns the hash code for the specified key.

```
protected virtual int GetHash (object key);  
  
abstract member GetHash : obj -> int  
override this.GetHash : obj -> int
```

## Parameters

key Object Object

The [Object](#) for which a hash code is to be returned.

## Returns

[Int32](#) [Int32](#)

The hash code for `key`.

## Exceptions

[NullReferenceException](#) [NullReferenceException](#)

`key` is `null`.

## Remarks

If the hash table was created with a specific [IHashCodeProvider](#) implementation, this method uses that hash code provider; otherwise, it uses the [Object.GetHashCode](#) implementation of `key`.

This method is an O(1) operation.

## See

[GetHashCode\(\)](#)[GetHashCode\(\)](#)

## Also

[Object](#)[Object](#)

[IHashCodeProvider](#)[IHashCodeProvider](#)

# Hashtable.GetObjectData Hashtable.GetObjectData

## In this Article

Implements the [ISerializable](#) interface and returns the data needed to serialize the [Hashtable](#).

```
[System.Security.SecurityCritical]
public virtual void GetObjectData (System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);

abstract member GetObjectData : System.Runtime.Serialization.SerializationInfo *
System.Runtime.Serialization.StreamingContext -> unit
override this.GetObjectData : System.Runtime.Serialization.SerializationInfo *
System.Runtime.Serialization.StreamingContext -> unit
```

## Parameters

info [SerializationInfo](#) [SerializationInfo](#)

A [SerializationInfo](#) object containing the information required to serialize the [Hashtable](#).

context [StreamingContext](#) [StreamingContext](#)

A [StreamingContext](#) object containing the source and destination of the serialized stream associated with the [Hashtable](#).

Attributes [SecurityCriticalAttribute](#)

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`info` is `null`.

[InvalidOperationException](#) [InvalidOperationException](#)

The collection was modified.

## Remarks

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

See

[ISerializable](#)  
[Serializable](#)

Also

[SerializationInfo](#)  
[SerializationInfo](#)

[StreamingContext](#)  
[StreamingContext](#)

[OnDeserialization\(Object\)](#)  
[OnDeserialization\(Object\)](#)

# Hashtable Hashtable

## In this Article

## Overloads

<code>Hashtable()</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the default initial capacity, load factor, hash code provider, and comparer.
<code>Hashtable(Int32, Single, IHashCodeProvider, IComparer)</code> <code>Hashtable(Int32, Single, IHashCodeProvider, IComparer)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the specified initial capacity, load factor, hash code provider, and comparer.
<code>Hashtable(IDictionary, Single, IHashCodeProvider, IComparer)</code> <code>Hashtable(IDictionary, Single, IHashCodeProvider, IComparer)</code>	Initializes a new instance of the <a href="#">Hashtable</a> class by copying the elements from the specified dictionary to the new <a href="#">Hashtable</a> object. The new <a href="#">Hashtable</a> object has an initial capacity equal to the number of elements copied, and uses the specified load factor, hash code provider, and comparer.
<code>Hashtable(Int32, Single, IEqualityComparer)</code> <code>Hashtable(Int32, Single, IEqualityComparer)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the specified initial capacity, load factor, and <a href="#">IEqualityComparer</a> object.
<code>Hashtable(Int32, IHashCodeProvider, IComparer)</code> <code>Hashtable(Int32, IHashCodeProvider, IComparer)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the specified initial capacity, hash code provider, comparer, and the default load factor.
<code>Hashtable(IDictionary, Single, IEqualityComparer)</code> <code>Hashtable(IDictionary, Single, IEqualityComparer)</code>	Initializes a new instance of the <a href="#">Hashtable</a> class by copying the elements from the specified dictionary to the new <a href="#">Hashtable</a> object. The new <a href="#">Hashtable</a> object has an initial capacity equal to the number of elements copied, and uses the specified load factor and <a href="#">IEqualityComparer</a> object.
<code>Hashtable(IDictionary, IHashCodeProvider, IComparer)</code> <code>Hashtable(IDictionary, IHashCodeProvider, IComparer)</code>	Initializes a new instance of the <a href="#">Hashtable</a> class by copying the elements from the specified dictionary to the new <a href="#">Hashtable</a> object. The new <a href="#">Hashtable</a> object has an initial capacity equal to the number of elements copied, and uses the default load factor, and the specified hash code provider and comparer. This API is obsolete. For an alternative, see <a href="#">Hashtable(IDictionary, IEqualityComparer)</a> .
<code>Hashtable(Int32, Single)</code> <code>Hashtable(Int32, Single)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the specified initial capacity and load factor, and the default hash code provider and comparer.

<code>Hashtable(SerializationInfo, StreamingContext) Hashtable(SerializationInfo, StreamingContext)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class that is serializable using the specified <a href="#">SerializationInfo</a> and <a href="#">StreamingContext</a> objects.
<code>Hashtable(IHashCodeProvider, IComparer) Hashtable(IHashCodeProvider, IComparer)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the default initial capacity and load factor, and the specified hash code provider and comparer.
<code>Hashtable(IDictionary, Single) Hashtable(IDictionary, Single)</code>	Initializes a new instance of the <a href="#">Hashtable</a> class by copying the elements from the specified dictionary to the new <a href="#">Hashtable</a> object. The new <a href="#">Hashtable</a> object has an initial capacity equal to the number of elements copied, and uses the specified load factor, and the default hash code provider and comparer.
<code>Hashtable(IDictionary, IEqualityComparer) Hashtable(IDictionary, IEqualityComparer)</code>	Initializes a new instance of the <a href="#">Hashtable</a> class by copying the elements from the specified dictionary to a new <a href="#">Hashtable</a> object. The new <a href="#">Hashtable</a> object has an initial capacity equal to the number of elements copied, and uses the default load factor and the specified <a href="#">IEqualityComparer</a> object.
<code>Hashtable(Int32) Hashtable(Int32)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the specified initial capacity, and the default load factor, hash code provider, and comparer.
<code>Hashtable(IEqualityComparer) Hashtable(IEqualityComparer)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the default initial capacity and load factor, and the specified <a href="#">IEqualityComparer</a> object.
<code>Hashtable(IDictionary) Hashtable(IDictionary)</code>	Initializes a new instance of the <a href="#">Hashtable</a> class by copying the elements from the specified dictionary to the new <a href="#">Hashtable</a> object. The new <a href="#">Hashtable</a> object has an initial capacity equal to the number of elements copied, and uses the default load factor, hash code provider, and comparer.
<code>Hashtable(Int32, IEqualityComparer) Hashtable(Int32, IEqualityComparer)</code>	Initializes a new, empty instance of the <a href="#">Hashtable</a> class using the specified initial capacity and <a href="#">IEqualityComparer</a> , and the default load factor.

## Hashtable()

Initializes a new, empty instance of the [Hashtable](#) class using the default initial capacity, load factor, hash code provider, and comparer.

```
public Hashtable();
```

### Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myComparer : IEqualityComparer
{
    public new bool Equals(object x, object y)
    {
        return x.Equals(y);
    }

    public int GetHashCode(object obj)
    {
        return obj.ToString().ToLower().GetHashCode();
    }
}

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a hash table using the specified IEqualityComparer that uses
    }
}
```

```

// the default Object.Equals to determine equality.
Hashtable myHT2 = new Hashtable(new myComparer());
myHT2.Add("FIRST", "Hello");
myHT2.Add("SECOND", "World");
myHT2.Add("THIRD", "!");

// Create a hash table using a case-insensitive hash code provider and
// case-insensitive comparer based on the InvariantCulture.
Hashtable myHT3 = new Hashtable(
    CaseInsensitiveHashCodeProvider.DefaultInvariant,
    CaseInsensitiveComparer.DefaultInvariant);
myHT3.Add("FIRST", "Hello");
myHT3.Add("SECOND", "World");
myHT3.Add("THIRD", "!");

// Create a hash table using an IEqualityComparer that is based on
// the Turkish culture (tr-TR) where "I" is not the uppercase
// version of "i".
CultureInfo myCul = new CultureInfo("tr-TR");
Hashtable myHT4 = new Hashtable(new myCultureComparer(myCul));
myHT4.Add("FIRST", "Hello");
myHT4.Add("SECOND", "World");
myHT4.Add("THIRD", "!");

// Search for a key in each hash table.
Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));
Console.WriteLine("first is in myHT4: {0}", myHT4.ContainsKey("first"));

}

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: False
first is in myHT3: True
first is in myHT4: False

*/

```

## Remarks

A hash table's capacity is used to calculate the optimal number of hash table buckets based on the load factor. Capacity is automatically increased as required.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#) object. The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

This constructor is an O(1) operation.

See

[GetHashCode\(\)](#)  
[GetHashCode\(\)](#)

Also

[Equals\(Object\)](#)  
[Equals\(Object\)](#)

## Hashtable(Int32, Single, IHashCodeProvider, IComparer) Hashtable(Int32, Single, IHashCodeProvider, IComparer)

Initializes a new, empty instance of the [Hashtable](#) class using the specified initial capacity, load factor, hash code provider, and comparer.

```
[System.Obsolete("Please use Hashtable(int, float, IEqualityComparer) instead.")]
public Hashtable (int capacity, float loadFactor, System.Collections.IHashCodeProvider hcp,
System.Collections.IComparer comparer);

new System.Collections.Hashtable : int * single * System.Collections.IHashCodeProvider *
System.Collections.IComparer -> System.Collections.Hashtable
```

Parameters

capacity

Int32 Int32

The approximate number of elements that the [Hashtable](#) object can initially contain.

loadFactor

Single Single

A number in the range from 0.1 through 1.0 that is multiplied by the default value which provides the best performance. The result is the maximum ratio of elements to buckets.

hcp

IHashCodeProvider IHashCodeProvider

The [IHashCodeProvider](#) object that supplies the hash codes for all keys in the [Hashtable](#).

-or-

`null` to use the default hash code provider, which is each key's implementation of [GetHashCode\(\)](#).

comparer

IComparer IComparer

The [IComparer](#) object to use to determine whether two keys are equal.

-or-

`null` to use the default comparer, which is each key's implementation of [Equals\(Object\)](#).

Attributes

[ObsoleteAttribute](#)

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

-or-

`loadFactor` is less than 0.1.

-or-

`loadFactor` is greater than 1.0.

Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(3, .8f);
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(3, .8f, new myCultureComparer());
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT3 = new Hashtable(3, .8f, new myCultureComparer(myCul));
    }
}
```

```

myHT3.Add("FIRST", "Hello");
myHT3.Add("SECOND", "World");
myHT3.Add("THIRD", "!");

// Search for a key in each hash table.
Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

}

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

Specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Hashtable](#) object. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption. A load factor of 1.0 is the best balance between speed and size.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The custom hash code provider and the custom comparer enable scenarios such as doing lookups with case-insensitive strings.

This constructor is an  $O(n)$  operation, where  $n$  is the [capacity](#) parameter.

See

[IHashCodeProvider](#)  
[IHashCodeProvider](#)

Also

[IComparer](#)  
[IComparer](#)  
[GetHashCode\(\)](#)  
[GetHashCode\(\)](#)  
[Equals\(Object\)](#)  
[Equals\(Object\)](#)

## Hashtable(IDictionary, Single, IHashCodeProvider, IComparer) Hashtable(IDictionary, Single, IHashCodeProvider, IComparer)

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the specified load factor, hash code provider, and comparer.

```
[System.Obsolete("Please use Hashtable(IDictionary, float, IEqualityComparer) instead.")]
public Hashtable (System.Collections.IDictionary d, float loadFactor,
System.Collections.IHashCodeProvider hcp, System.Collections.IComparer comparer);

new System.Collections.Hashtable : System.Collections.IDictionary * single *
System.Collections.IHashCodeProvider * System.Collections.IComparer -> System.Collections.Hashtable
```

## Parameters

d IDictionary IDictionary

The [IDictionary](#) object to copy to a new [Hashtable](#) object.

loadFactor Single Single

A number in the range from 0.1 through 1.0 that is multiplied by the default value which provides the best performance. The result is the maximum ratio of elements to buckets.

hcp IHashCodeProvider IHashCodeProvider

The [IHashCodeProvider](#) object that supplies the hash codes for all keys in the [Hashtable](#).

-or-

`null` to use the default hash code provider, which is each key's implementation of [GetHashCode\(\)](#).

comparer IComparer IComparer

The [IComparer](#) object to use to determine whether two keys are equal.

-or-

`null` to use the default comparer, which is each key's implementation of [Equals\(Object\)](#).

Attributes ObsoleteAttribute

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`d` is `null`.

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`loadFactor` is less than 0.1.

-or-

`loadFactor` is greater than 1.0.

## Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = new CaseInsensitiveComparer();
    }

    public int GetHashCode(object obj)
    {
        return myComparer.GetHashCode(obj);
    }

    public bool Equals(object x, object y)
    {
        return myComparer.Equals(x, y);
    }
}
```

```

{
    myComparer = CaseInsensitiveComparer.DefaultInvariant;
}

public myCultureComparer(CultureInfo myCulture)
{
    myComparer = new CaseInsensitiveComparer(myCulture);
}

public new bool Equals(object x, object y)
{
    if (myComparer.Compare(x, y) == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

public int GetHashCode(object obj)
{
    // Compare the hash code for the lowercase versions of the strings.
    return obj.ToString().ToLower().GetHashCode();
}
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create the dictionary.
        SortedList mySL = new SortedList();
        mySL.Add("FIRST", "Hello");
        mySL.Add("SECOND", "World");
        mySL.Add("THIRD", "!");

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(mySL, .8f);

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(mySL, .8f,
            new myCultureComparer());

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT3 = new Hashtable(mySL, .8f, new myCultureComparer(myCul));

        // Search for a key in each hash table.
        Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
        Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
        Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

    }
}

/*
This code produces the following output.

```

```
... produces the following output.  
Results vary depending on the system's culture settings.
```

```
first is in myHT1: False  
first is in myHT2: True  
first is in myHT3: False  
  
*/
```

## Remarks

The initial capacity is set to the number of elements in the source dictionary. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption. A load factor of 1.0 is the best balance between speed and size.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#) object. The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The custom hash code provider and the custom comparer enable scenarios such as doing lookups with case-insensitive strings.

The elements of the new [Hashtable](#) are sorted in the same order in which the enumerator iterates through the [IDictionary](#) object.

This constructor is an O( $n$ ) operation, where  $n$  is the number of elements in the  $d$  parameter.

## Hashtable(Int32, Single, IEqualityComparer) Hashtable(Int32, Single, IEqualityComparer)

Initializes a new, empty instance of the [Hashtable](#) class using the specified initial capacity, load factor, and [IEqualityComparer](#) object.

```
public Hashtable (int capacity, float loadFactor, System.Collections.IEqualityComparer  
equalityComparer);  
  
new System.Collections.Hashtable : int * single * System.Collections.IEqualityComparer ->  
System.Collections.Hashtable
```

### Parameters

capacity	<a href="#">Int32</a>	<a href="#">Int32</a>
----------	-----------------------	-----------------------

The approximate number of elements that the [Hashtable](#) object can initially contain.

loadFactor	<a href="#">Single</a>	<a href="#">Single</a>
------------	------------------------	------------------------

A number in the range from 0.1 through 1.0 that is multiplied by the default value which provides the best performance. The result is the maximum ratio of elements to buckets.

equalityComparer	<a href="#">IEqualityComparer</a>	<a href="#">IEqualityComparer</a>
------------------	-----------------------------------	-----------------------------------

The [IEqualityComparer](#) object that defines the hash code provider and the comparer to use with the [Hashtable](#).

-or-

`null` to use the default hash code provider and the default comparer. The default hash code provider is each key's implementation of [GetHashCode\(\)](#) and the default comparer is each key's implementation of [Equals\(Object\)](#).

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

-or-

`loadFactor` is less than 0.1.

-or-

`loadFactor` is greater than 1.0.

## Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
}
```

```

{
    // Create a hash table using the default comparer.
    Hashtable myHT1 = new Hashtable(3, .8f);
    myHT1.Add("FIRST", "Hello");
    myHT1.Add("SECOND", "World");
    myHT1.Add("THIRD", "!");

    // Create a hash table using the specified IEqualityComparer that uses
    // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
    Hashtable myHT2 = new Hashtable(3, .8f, new myCultureComparer());
    myHT2.Add("FIRST", "Hello");
    myHT2.Add("SECOND", "World");
    myHT2.Add("THIRD", "!");

    // Create a hash table using an IEqualityComparer that is based on
    // the Turkish culture (tr-TR) where "I" is not the uppercase
    // version of "i".
    CultureInfo myCul = new CultureInfo("tr-TR");
    Hashtable myHT3 = new Hashtable(3, .8f, new myCultureComparer(myCul));

    myHT3.Add("FIRST", "Hello");
    myHT3.Add("SECOND", "World");
    myHT3.Add("THIRD", "!");

    // Search for a key in each hash table.
    Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
    Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
    Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

}
}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

Specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Hashtable](#) object. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption. A load factor of 1.0 is the best balance between speed and size.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The [IEqualityComparer](#) object includes both the hash code provider and the comparer. If an [IEqualityComparer](#) is used in the [Hashtable](#) constructor, the objects used as keys in the [Hashtable](#) are not required to override the [Object.GetHashCode](#) and [Object.Equals](#) methods.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The [IEqualityComparer](#) enables scenarios such as doing lookups with case-insensitive strings.

This constructor is an O( $n$ ) operation, where  $n$  is the `capacity` parameter.

See

[IEqualityComparer](#)[IEqualityComparer](#)

Also

## Hashtable(Int32, IHashCodeProvider, IComparer) Hashtable(Int32, IHashCodeProvider, IComparer)

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

```
[System.Obsolete("Please use Hashtable(int, IEqualityComparer) instead.")]
public Hashtable (int capacity, System.Collections.IHashCodeProvider hcp,
System.Collections.IComparer comparer);

new System.Collections.Hashtable : int * System.Collections.IHashCodeProvider *
System.Collections.IComparer -> System.Collections.Hashtable
```

Parameters

capacity

Int32 Int32

The approximate number of elements that the [Hashtable](#) object can initially contain.

hcp

IHashCodeProvider IHashCodeProvider

The [IHashCodeProvider](#) object that supplies the hash codes for all keys in the [Hashtable](#).

-or-

`null` to use the default hash code provider, which is each key's implementation of [GetHashCode\(\)](#).

comparer

IComparer IComparer

The [IComparer](#) object to use to determine whether two keys are equal.

-or-

`null` to use the default comparer, which is each key's implementation of [Equals\(Object\)](#).

Attributes

ObsoleteAttribute

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;
```

```

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(3);
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(3, new myCultureComparer());
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT3 = new Hashtable(3, new myCultureComparer(myCul));
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Search for a key in each hash table.
        Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
        Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
        Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));
    }
}

```

```

    }

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

Specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Hashtable](#) object. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The custom hash code provider and the custom comparer enable scenarios such as doing lookups with case-insensitive strings.

This constructor is an O( $n$ ) operation, where  $n$  is the [capacity](#) parameter.

See

[IHashCodeProvider](#)  
[IEqualityComparer](#)

Also

[GetHashCode\(\)](#)  
[Equals\(Object\)](#)

## **Hashtable(IDictionary, Single, IEqualityComparer)**

## **Hashtable(IDictionary, Single, IEqualityComparer)**

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the specified load factor and [IEqualityComparer](#) object.

```

public Hashtable (System.Collections.IDictionary d, float loadFactor,
System.Collections.IEqualityComparer equalityComparer);

new System.Collections.Hashtable : System.Collections.IDictionary * single *
System.Collections.IEqualityComparer -> System.Collections.Hashtable

```

## Parameters

d	<a href="#">IDictionary</a>
---	-----------------------------

The [IDictionary](#) object to copy to a new [Hashtable](#) object.

loadFactor

Single Single

A number in the range from 0.1 through 1.0 that is multiplied by the default value which provides the best performance. The result is the maximum ratio of elements to buckets.

equalityComparer

[IEqualityComparer](#) [IEqualityComparer](#)

The [IEqualityComparer](#) object that defines the hash code provider and the comparer to use with the [Hashtable](#).

-or-

`null` to use the default hash code provider and the default comparer. The default hash code provider is each key's implementation of [GetHashCode\(\)](#) and the default comparer is each key's implementation of [Equals\(Object\)](#).

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`d` is `null`.

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`loadFactor` is less than 0.1.

-or-

`loadFactor` is greater than 1.0.

## Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

```

public int GetHashCode(object obj)
{
    // Compare the hash code for the lowercase versions of the strings.
    return obj.ToString().ToLower().GetHashCode();
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create the dictionary.
        SortedList mySL = new SortedList();
        mySL.Add("FIRST", "Hello");
        mySL.Add("SECOND", "World");
        mySL.Add("THIRD", "!");

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(mySL, .8f);

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(mySL, .8f,
            new myCultureComparer());

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT3 = new Hashtable(mySL, .8f, new myCultureComparer(myCul));

        // Search for a key in each hash table.
        Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
        Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
        Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

    }
}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

The initial capacity is set to the number of elements in the source dictionary. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption. A load factor of 1.0 is the best balance between speed and size.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the

smallest prime number that is larger than twice the current number of buckets.

The [IEqualityComparer](#) object includes both the hash code provider and the comparer. If an [IEqualityComparer](#) is used in the [Hashtable](#) constructor, the objects used as keys in the [Hashtable](#) object are not required to override the [Object.GetHashCode](#) and [Object.Equals](#) methods.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The [IEqualityComparer](#) enables scenarios such as doing lookups with case-insensitive strings.

The elements of the new [Hashtable](#) are sorted in the same order in which the enumerator iterates through the [IDictionary](#) object.

This constructor is an  $O(n)$  operation, where  $n$  is the number of elements in the  $d$  parameter.

See

[IEqualityComparer](#)

Also

## **Hashtable(IDictionary, IHashCodeProvider, IComparer)**

## **Hashtable(IDictionary, IHashCodeProvider, IComparer)**

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the default load factor, and the specified hash code provider and comparer. This API is obsolete. For an alternative, see [Hashtable\(IDictionary, IEqualityComparer\)](#).

```
[System.Obsolete("Please use Hashtable(IDictionary, IEqualityComparer) instead.")]
public Hashtable (System.Collections.IDictionary d, System.Collections.IHashCodeProvider hcp,
System.Collections.IComparer comparer);

new System.Collections.Hashtable : System.Collections.IDictionary *
System.Collections.IHashCodeProvider * System.Collections.IComparer -> System.Collections.Hashtable
```

Parameters

**d** [IDictionary](#) [IDictionary](#)

The [IDictionary](#) object to copy to a new [Hashtable](#) object.

**hcp** [IHashCodeProvider](#) [IHashCodeProvider](#)

The [IHashCodeProvider](#) object that supplies the hash codes for all keys in the [Hashtable](#).

-or-

**null** to use the default hash code provider, which is each key's implementation of [GetHashCode\(\)](#).

**comparer** [IComparer](#) [IComparer](#)

The [IComparer](#) object to use to determine whether two keys are equal.

-or-

**null** to use the default comparer, which is each key's implementation of [Equals\(Object\)](#).

Attributes

[ObsoleteAttribute](#)

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

d is null.

## Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create the dictionary.
        SortedList mySL = new SortedList();
        mySL.Add("FIRST", "Hello");
        mySL.Add("SECOND", "World");
        mySL.Add("THIRD", "!");

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(mySL);

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(mySL, new myCultureComparer());
    }
}
```

```

// Create a hash table using an IEqualityComparer that is based on
// the Turkish culture (tr-TR) where "I" is not the uppercase
// version of "i".
CultureInfo myCul = new CultureInfo("tr-TR");
Hashtable myHT3 = new Hashtable(mySL, new myCultureComparer(myCul));

// Search for a key in each hash table.
Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

The initial capacity is set to the number of elements in the source dictionary. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#) object. The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The custom hash code provider and the custom comparer enable scenarios such as doing lookups with case-insensitive strings.

The elements of the new [Hashtable](#) are sorted in the same order in which the enumerator iterates through the [IDictionary](#) object.

This constructor is an O( $n$ ) operation, where  $n$  is the number of elements in the  $d$  parameter.

See

[IDictionary](#)  
[IDictionary](#)

Also

[IHashCodeProvider](#)  
[IHashCodeProvider](#)  
[IComparer](#)  
[IComparer](#)  
[GetHashCode\(\)](#)  
[GetHashCode\(\)](#)  
[Equals\(Object\)](#)  
[Equals\(Object\)](#)

## Hashtable(Int32, Single) Hashtable(Int32, Single)

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

```
public Hashtable (int capacity, float loadFactor);
new System.Collections.Hashtable : int * single -> System.Collections.Hashtable
```

## Parameters

capacity Int32 Int32

The approximate number of elements that the [Hashtable](#) object can initially contain.

loadFactor Single Single

A number in the range from 0.1 through 1.0 that is multiplied by the default value which provides the best performance. The result is the maximum ratio of elements to buckets.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentException](#)

`capacity` is less than zero.

-or-

`loadFactor` is less than 0.1.

-or-

`loadFactor` is greater than 1.0.

[ArgumentException](#) [ArgumentException](#)

`capacity` is causing an overflow.

## Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        return myComparer.GetHashCode(obj);
    }
}
```

```

        }
    else
    {
        return false;
    }
}

public int GetHashCode(object obj)
{
    // Compare the hash code for the lowercase versions of the strings.
    return obj.ToString().ToLower().GetHashCode();
}
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(3, .8f);
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(3, .8f, new myCultureComparer());
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT3 = new Hashtable(3, .8f, new myCultureComparer(myCul));

        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Search for a key in each hash table.
        Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
        Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
        Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

    }
}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False
*/

```

## Remarks

Specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Hashtable](#) object. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption. A load factor of 1.0 is the best balance between speed and size.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

This constructor is an  $O(n)$  operation, where  $n$  is the `capacity` parameter.

See

[GetHashCode\(\)](#)  
[Equals\(Object\)](#)

Also

[GetHashCode\(\)](#)  
[Equals\(Object\)](#)

## **Hashtable(SerializationInfo, StreamingContext)**

## **Hashtable(SerializationInfo, StreamingContext)**

Initializes a new, empty instance of the [Hashtable](#) class that is serializable using the specified [SerializationInfo](#) and [StreamingContext](#) objects.

```
protected Hashtable (System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);

new System.Collections.Hashtable : System.Runtime.Serialization.SerializationInfo *
System.Runtime.Serialization.StreamingContext -> System.Collections.Hashtable
```

Parameters

info

[SerializationInfo](#) [SerializationInfo](#)

A [SerializationInfo](#) object containing the information required to serialize the [Hashtable](#) object.

context

[StreamingContext](#) [StreamingContext](#)

A [StreamingContext](#) object containing the source and destination of the serialized stream associated with the [Hashtable](#).

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`info` is `null`.

Remarks

A hash table's capacity is used to calculate the optimal number of hash table buckets based on the load factor. Capacity is automatically increased as required.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#) object. The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

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

Because serializing and deserializing an enumerator for a [Hashtable](#) can cause the elements to become reordered, it is not possible to continue enumeration without calling the [Reset](#) method.

See

[ISerializable](#)  
[Serializable](#)

Also

[SerializationInfo](#)  
[SerializationInfo](#)  
[StreamingContext](#)  
[StreamingContext](#)  
[OnDeserialization\(Object\)](#)  
[OnDeserialization\(Object\)](#)  
[GetHashCode\(\)](#)  
[Equals\(Object\)](#)  
[Equals\(Object\)](#)

## **Hashtable(IHashCodeProvider, IComparer)**

## **Hashtable(IHashCodeProvider, IComparer)**

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

```
[System.Obsolete("Please use Hashtable(IEqualityComparer) instead.")]
public Hashtable (System.Collections.IHashCodeProvider hcp, System.Collections.IComparer comparer);
new System.Collections.Hashtable : System.Collections.IHashCodeProvider *
System.Collections.IComparer -> System.Collections.Hashtable
```

Parameters

hcp

[IHashCodeProvider](#) [IHashCodeProvider](#)

The [IHashCodeProvider](#) object that supplies the hash codes for all keys in the [Hashtable](#) object.

-or-

`null` to use the default hash code provider, which is each key's implementation of [GetHashCode\(\)](#).

comparer

[IComparer](#) [IComparer](#)

The [IComparer](#) object to use to determine whether two keys are equal.

-or-

`null` to use the default comparer, which is each key's implementation of [Equals\(Object\)](#).

Attributes

[ObsoleteAttribute](#)

Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myComparer : IEqualityComparer
```

```

{
    public new bool Equals(object x, object y)
    {
        return x.Equals(y);
    }

    public int GetHashCode(object obj)
    {
        return obj.ToString().ToLower().GetHashCode();
    }
}

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a hash table using the specified IEqualityComparer that uses
        // the default Object.Equals to determine equality.
        Hashtable myHT2 = new Hashtable(new myComparer());
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a hash table using a case-insensitive hash code provider and
        // case-insensitive comparer based on the InvariantCulture.
        Hashtable mvHT3 = new Hashtable(

```

```

        CaseInsensitiveHashCodeProvider.DefaultInvariant,
        CaseInsensitiveComparer.DefaultInvariant);
myHT3.Add("FIRST", "Hello");
myHT3.Add("SECOND", "World");
myHT3.Add("THIRD", "!");

// Create a hash table using an IEqualityComparer that is based on
// the Turkish culture (tr-TR) where "I" is not the uppercase
// version of "i".
CultureInfo myCul = new CultureInfo("tr-TR");
Hashtable myHT4 = new Hashtable(new myCultureComparer(myCul));
myHT4.Add("FIRST", "Hello");
myHT4.Add("SECOND", "World");
myHT4.Add("THIRD", "!");

// Search for a key in each hash table.
Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));
Console.WriteLine("first is in myHT4: {0}", myHT4.ContainsKey("first"));

}

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: False
first is in myHT3: True
first is in myHT4: False

*/

```

## Remarks

A hash table's capacity is used to calculate the optimal number of hash table buckets based on the load factor. Capacity is automatically increased as required.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#) object. The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The custom hash code provider and the custom comparer enable scenarios such as doing lookups with case-insensitive strings.

This constructor is an O(1) operation.

See

[IHashCodeProvider](#)  
[IHashCodeProvider](#)

Also

[IComparer](#)  
[IComparer](#)

## Hashtable(IDictionary, Single) Hashtable(IDictionary, Single)

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the specified load factor, and the default hash code provider and comparer.

```
public Hashtable (System.Collections.IDictionary d, float loadFactor);  
new System.Collections.Hashtable : System.Collections.IDictionary * single ->  
System.Collections.Hashtable
```

### Parameters

d [IDictionary](#) [IDictionary](#)

The [IDictionary](#) object to copy to a new [Hashtable](#) object.

loadFactor [Single](#) [Single](#)

A number in the range from 0.1 through 1.0 that is multiplied by the default value which provides the best performance. The result is the maximum ratio of elements to buckets.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`d` is `null`.

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`loadFactor` is less than 0.1.

-or-

`loadFactor` is greater than 1.0.

### Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;  
using System.Collections;  
using System.Globalization;  
  
class myCultureComparer : IEqualityComparer  
{  
    public CaseInsensitiveComparer myComparer;  
  
    public myCultureComparer()  
    {  
        myComparer = CaseInsensitiveComparer.DefaultInvariant;  
    }  
  
    public myCultureComparer(CultureInfo myCulture)  
    {  
        myComparer = new CaseInsensitiveComparer(myCulture);  
    }  
  
    public new bool Equals(object x, object y)
```

```

{
    if (myComparer.Compare(x, y) == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

public int GetHashCode(object obj)
{
    // Compare the hash code for the lowercase versions of the strings.
    return obj.ToString().ToLower().GetHashCode();
}
}

public class SamplesHashtable
{
    public static void Main()
    {

        // Create the dictionary.
        SortedList mySL = new SortedList();
        mySL.Add("FIRST", "Hello");
        mySL.Add("SECOND", "World");
        mySL.Add("THIRD", "!");

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(mySL, .8f);

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(mySL, .8f,
            new myCultureComparer());

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT3 = new Hashtable(mySL, .8f, new myCultureComparer(myCul));

        // Search for a key in each hash table.
        Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
        Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
        Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

    }
}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False
*/

```

## Remarks

The initial capacity is set to the number of elements in the source dictionary. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption. A load factor of 1.0 is the best balance between speed and size.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#) object. The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The elements of the new [Hashtable](#) are sorted in the same order in which the enumerator iterates through the [IDictionary](#) object.

This constructor is an O( $n$ ) operation, where  $n$  is the number of elements in the  $d$  parameter.

See

[IDictionary](#)

Also

[GetHashCode\(\)](#)

[Equals\(Object\)](#)

[Equals\(Object\)](#)

## Hashtable(IDictionary, IEqualityComparer) Hashtable(IDictionary, IEqualityComparer)

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to a new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the default load factor and the specified [IEqualityComparer](#) object.

```
public Hashtable (System.Collections.IDictionary d, System.Collections.IEqualityComparer  
equalityComparer);  
  
new System.Collections.Hashtable : System.Collections.IDictionary *  
System.Collections.IEqualityComparer -> System.Collections.Hashtable
```

### Parameters

$d$  [IDictionary](#) [IDictionary](#)

The [IDictionary](#) object to copy to a new [Hashtable](#) object.

$equalityComparer$  [IEqualityComparer](#) [IEqualityComparer](#)

The [IEqualityComparer](#) object that defines the hash code provider and the comparer to use with the [Hashtable](#).

-or-

$null$  to use the default hash code provider and the default comparer. The default hash code provider is each key's implementation of [GetHashCode\(\)](#) and the default comparer is each key's implementation of [Equals\(Object\)](#).

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

$d$  is  $null$ .

## Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create the dictionary.
        SortedList mySL = new SortedList();
        mySL.Add("FIRST", "Hello");
        mySL.Add("SECOND", "World");
        mySL.Add("THIRD", "!");

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(mySL);

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(mySL, new myCultureComparer());

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
```

```

Hashtable myHT3 = new Hashtable(mySL, new myCultureComparer(myCul));

// Search for a key in each hash table.
Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

}

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

The initial capacity is set to the number of elements in the source dictionary. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The [IEqualityComparer](#) object includes both the hash code provider and the comparer. If an [IEqualityComparer](#) is used in the [Hashtable](#) constructor, the objects used as keys in the [Hashtable](#) object are not required to override the [Object.GetHashCode](#) and [Object.Equals](#) methods.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The [IEqualityComparer](#) enables scenarios such as doing lookups with case-insensitive strings.

The elements of the new [Hashtable](#) are sorted in the same order in which the enumerator iterates through the [IDictionary](#) object.

This constructor is an  $O(n)$  operation, where  $n$  is the number of elements in the  $d$  parameter.

See

[IDictionary](#)

Also

[IEqualityComparer](#)

## Hashtable(Int32) Hashtable(Int32)

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

```
public Hashtable (int capacity);
new System.Collections.Hashtable : int -> System.Collections.Hashtable
```

## Parameters

capacity Int32 Int32

The approximate number of elements that the [Hashtable](#) object can initially contain.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

## Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(3);
```

```

Hashtable myHT1 = new Hashtable(3,
    myHT1.Add("FIRST", "Hello");
    myHT1.Add("SECOND", "World");
    myHT1.Add("THIRD", "!"));

    // Create a hash table using the specified IEqualityComparer that uses
    // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
    Hashtable myHT2 = new Hashtable(3, new myCultureComparer());
    myHT2.Add("FIRST", "Hello");
    myHT2.Add("SECOND", "World");
    myHT2.Add("THIRD", "!");

    // Create a hash table using an IEqualityComparer that is based on
    // the Turkish culture (tr-TR) where "I" is not the uppercase
    // version of "i".
    CultureInfo myCul = new CultureInfo("tr-TR");
    Hashtable myHT3 = new Hashtable(3, new myCultureComparer(myCul));
    myHT3.Add("FIRST", "Hello");
    myHT3.Add("SECOND", "World");
    myHT3.Add("THIRD", "!");

    // Search for a key in each hash table.
    Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
    Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
    Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

}

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

Specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Hashtable](#) object. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

This constructor is an O( $n$ ) operation, where  $n$  is [capacity](#).

See

[GetHashCode\(\)](#)  
[GetHashCode\(\)](#)

Also

[Equals\(Object\)](#)  
[Equals\(Object\)](#)

## Hashtable(IEqualityComparer) Hashtable(IEqualityComparer)

Initializes a new, empty instance of the [Hashtable](#) class using the default initial capacity and load factor, and the specified [IEqualityComparer](#) object.

```
public Hashtable (System.Collections.IEqualityComparer equalityComparer);  
new System.Collections.Hashtable : System.Collections.IEqualityComparer ->  
System.Collections.Hashtable
```

Parameters

equalityComparer [IEqualityComparer](#) [IEqualityComparer](#)

The [IEqualityComparer](#) object that defines the hash code provider and the comparer to use with the [Hashtable](#) object.

-or-

`null` to use the default hash code provider and the default comparer. The default hash code provider is each key's implementation of [GetHashCode\(\)](#) and the default comparer is each key's implementation of [Equals\(Object\)](#).

### Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;  
using System.Collections;  
using System.Globalization;  
  
class myComparer : IEqualityComparer  
{  
    public new bool Equals(object x, object y)  
    {  
        return x.Equals(y);  
    }  
  
    public int GetHashCode(object obj)  
    {  
        return obj.ToString().ToLower().GetHashCode();  
    }  
}  
  
class myCultureComparer : IEqualityComparer  
{  
    public CaseInsensitiveComparer myComparer;  
  
    public myCultureComparer()  
    {  
        myComparer = CaseInsensitiveComparer.DefaultInvariant;  
    }  
  
    public myCultureComparer(CultureInfo myCulture)  
    {  
        myComparer = new CaseInsensitiveComparer(myCulture);  
    }  
  
    public new bool Equals(object x, object y)  
    {  
        if (myComparer.Compare(x, y) == 0)  
        {  
            return true;  
        }  
        else
```

```

    {
        return false;
    }
}

public int GetHashCode(object obj)
{
    return obj.ToString().ToLower().GetHashCode();
}
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable();
        myHT1.Add("FIRST", "Hello");
        myHT1.Add("SECOND", "World");
        myHT1.Add("THIRD", "!");

        // Create a hash table using the specified IEqualityComparer that uses
        // the default Object.Equals to determine equality.
        Hashtable myHT2 = new Hashtable(new myComparer());
        myHT2.Add("FIRST", "Hello");
        myHT2.Add("SECOND", "World");
        myHT2.Add("THIRD", "!");

        // Create a hash table using a case-insensitive hash code provider and
        // case-insensitive comparer based on the InvariantCulture.
        Hashtable myHT3 = new Hashtable(
            CaseInsensitiveHashCodeProvider.DefaultInvariant,
            CaseInsensitiveComparer.DefaultInvariant);
        myHT3.Add("FIRST", "Hello");
        myHT3.Add("SECOND", "World");
        myHT3.Add("THIRD", "!");

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT4 = new Hashtable(new myCultureComparer(myCul));
        myHT4.Add("FIRST", "Hello");
        myHT4.Add("SECOND", "World");
        myHT4.Add("THIRD", "!");

        // Search for a key in each hash table.
        Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
        Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
        Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));
        Console.WriteLine("first is in myHT4: {0}", myHT4.ContainsKey("first"));

    }
}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: False

```

```
first is in myHT3: True
first is in myHT4: False

*/
```

## Remarks

A hash table's capacity is used to calculate the optimal number of hash table buckets based on the load factor. Capacity is automatically increased as required.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The [IEqualityComparer](#) object includes both the hash code provider and the comparer. If an [IEqualityComparer](#) is used in the [Hashtable](#) constructor, the objects used as keys in the [Hashtable](#) object are not required to override the [Object.GetHashCode](#) and [Object.Equals](#) methods.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The [IEqualityComparer](#) enables scenarios such as doing lookups with case-insensitive strings.

This constructor is an O(1) operation.

See

[IEqualityComparer](#)[EqualityComparer](#)

Also

## Hashtable(IDictionary) Hashtable(IDictionary)

Initializes a new instance of the [Hashtable](#) class by copying the elements from the specified dictionary to the new [Hashtable](#) object. The new [Hashtable](#) object has an initial capacity equal to the number of elements copied, and uses the default load factor, hash code provider, and comparer.

```
public Hashtable (System.Collections.IDictionary d);
new System.Collections.Hashtable : System.Collections.IDictionary -> System.Collections.Hashtable
```

Parameters

d [IDictionary](#) [IDictionary](#)

The [IDictionary](#) object to copy to a new [Hashtable](#) object.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

d is null.

Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System:
```

```
-----,
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{

    public static void Main()
    {

        // Create the dictionary.
        SortedList mySL = new SortedList();
        mySL.Add("FIRST", "Hello");
        mySL.Add("SECOND", "World");
        mySL.Add("THIRD", "!");

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(mySL);

        // Create a hash table using the specified IEqualityComparer that uses
        // the CaseInsensitiveComparer.DefaultInvariant to determine equality.
        Hashtable myHT2 = new Hashtable(mySL, new myCultureComparer());

        // Create a hash table using an IEqualityComparer that is based on
        // the Turkish culture (tr-TR) where "I" is not the uppercase
        // version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        Hashtable myHT3 = new Hashtable(mySL, new myCultureComparer(myCul));

        // Search for a key in each hash table.
        Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
        Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
        Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));
    }
}
```

```

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

The initial capacity is set to the number of elements in the source dictionary. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The hash code provider dispenses hash codes for keys in the [Hashtable](#) object. The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The elements of the new [Hashtable](#) are sorted in the same order in which the enumerator iterates through the [IDictionary](#) object.

This constructor is an O( $n$ ) operation, where  $n$  is the number of elements in the  $d$  parameter.

See

[IDictionary](#)

Also

[GetHashCode](#)

[Equals](#)

## Hashtable(Int32, IEqualityComparer) Hashtable(Int32, IEqualityComparer)

Initializes a new, empty instance of the [Hashtable](#) class using the specified initial capacity and [IEqualityComparer](#), and the default load factor.

```

public Hashtable (int capacity, System.Collections.IEqualityComparer equalityComparer);
new System.Collections.Hashtable : int * System.Collections.IEqualityComparer ->
System.Collections.Hashtable

```

## Parameters

capacity

[Int32](#)

The approximate number of elements that the [Hashtable](#) object can initially contain.

equalityComparer

IEqualityComparer IEqualityComparer

The [IEqualityComparer](#) object that defines the hash code provider and the comparer to use with the [Hashtable](#).

-or-

`null` to use the default hash code provider and the default comparer. The default hash code provider is each key's implementation of [GetHashCode\(\)](#) and the default comparer is each key's implementation of [Equals\(Object\)](#).

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

Examples

The following code example creates hash tables using different [Hashtable](#) constructors and demonstrates the differences in the behavior of the hash tables, even if each one contains the same elements.

```
using System;
using System.Collections;
using System.Globalization;

class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        // Compare the hash code for the lowercase versions of the strings.
        return obj.ToString().ToLower().GetHashCode();
    }
}

public class SamplesHashtable
{
    public static void Main()
    {

        // Create a hash table using the default comparer.
        Hashtable myHT1 = new Hashtable(3);
```

```

myHT1.Add("FIRST", "Hello");
myHT1.Add("SECOND", "World");
myHT1.Add("THIRD", "!");

// Create a hash table using the specified IEqualityComparer that uses
// the CaseInsensitiveComparer.DefaultInvariant to determine equality.
Hashtable myHT2 = new Hashtable(3, new myCultureComparer());
myHT2.Add("FIRST", "Hello");
myHT2.Add("SECOND", "World");
myHT2.Add("THIRD", "!");

// Create a hash table using an IEqualityComparer that is based on
// the Turkish culture (tr-TR) where "I" is not the uppercase
// version of "i".
CultureInfo myCul = new CultureInfo("tr-TR");
Hashtable myHT3 = new Hashtable(3, new myCultureComparer(myCul));
myHT3.Add("FIRST", "Hello");
myHT3.Add("SECOND", "World");
myHT3.Add("THIRD", "!");

// Search for a key in each hash table.
Console.WriteLine("first is in myHT1: {0}", myHT1.ContainsKey("first"));
Console.WriteLine("first is in myHT2: {0}", myHT2.ContainsKey("first"));
Console.WriteLine("first is in myHT3: {0}", myHT3.ContainsKey("first"));

}

}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

first is in myHT1: False
first is in myHT2: True
first is in myHT3: False

*/

```

## Remarks

Specifying the initial capacity eliminates the need to perform a number of resizing operations while adding elements to the [Hashtable](#) object. Capacity is automatically increased as required based on the load factor.

The load factor is the maximum ratio of elements to buckets. A smaller load factor means faster lookup at the cost of increased memory consumption.

When the actual load factor reaches the specified load factor, the number of buckets is automatically increased to the smallest prime number that is larger than twice the current number of buckets.

The [IEqualityComparer](#) object includes both the hash code provider and the comparer. If an [IEqualityComparer](#) is used in the [Hashtable](#) constructor, the objects used as keys in the [Hashtable](#) are not required to override the [Object.GetHashCode](#) and [Object.Equals](#) methods.

The hash code provider dispenses hash codes for keys in the [Hashtable](#). The default hash code provider is the key's implementation of [Object.GetHashCode](#).

The comparer determines whether two keys are equal. Every key in a [Hashtable](#) must be unique. The default comparer is the key's implementation of [Object.Equals](#).

The [IEqualityComparer](#) enables scenarios such as doing lookups with case-insensitive strings.

This constructor is an  $O(n)$  operation, where  $n$  is the `capacity` parameter.

See

[IEqualityComparer](#)[IEqualityComparer](#)

Also

# Hashtable.hcp Hashtable.hcp

## In this Article

Gets or sets the object that can dispense hash codes.

```
[System.Obsolete("Please use EqualityComparer property.")]
protected System.Collections.IHashCodeProvider hcp { get; set; }

member this.hcp : System.Collections.IHashCodeProvider with get, set
```

Returns

[IHashCodeProvider](#) [IHashCodeProvider](#)

The object that can dispense hash codes.

Attributes

[ObsoleteAttribute](#)

Exceptions

[ArgumentException](#) [ArgumentException](#)

The property is set to a value, but the hash table was created using an [IEqualityComparer](#).

## Remarks

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

See

[IHashCodeProvider](#) [IHashCodeProvider](#)

Also

# Hashtable.IEnumerable.GetEnumerator

## In this Article

Returns an enumerator that iterates through a collection.

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

Returns

[IEnumerator](#)

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

## Remarks

[Visual Basic, C#]

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, calling [Current](#) throws an exception. 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`, calling [Current](#) throws an exception. 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 can be invalidated and the next call to [MoveNext](#) or [Reset](#) throws an [InvalidOperationException](#). If the collection is modified between [MoveNext](#) and [Current](#), [Current](#) returns the element that it is set to, even if the enumerator is already invalidated.

The enumerator does not have exclusive access to the collection; therefore, 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 causes 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.

This method is an O(1) operation.

See

[IEnumerator](#)

Also

# Hashtable.IsFixedSize Hashtable.IsFixedSize

## In this Article

Gets a value indicating whether the [Hashtable](#) has a fixed size.

```
public virtual bool IsFixedSize { get; }  
member this.IsFixedSize : bool
```

Returns

[Boolean](#)

`true` if the [Hashtable](#) has a fixed size; otherwise, `false`. The default is `false`.

## 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.

# Hashtable.IsReadOnly Hashtable.IsReadOnly

## In this Article

Gets a value indicating whether the [Hashtable](#) is read-only.

```
public virtual bool IsReadOnly { get; }  
member this.IsReadOnly : bool
```

Returns

[Boolean](#)

`true` if the [Hashtable](#) is read-only; otherwise, `false`. The default is `false`.

## 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.

# Hashtable.IsSynchronized Hashtable.IsSynchronized

## In this Article

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

```
public virtual bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean](#)

`true` if access to the [Hashtable](#) is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Examples

The following example shows how to synchronize a [Hashtable](#), determine if a [Hashtable](#) is synchronized, and use a synchronized [Hashtable](#).

```
using System;  
using System.Collections;  
public class SamplesHashtable {  
  
    public static void Main() {  
  
        // Creates and initializes a new Hashtable.  
        Hashtable myHT = new Hashtable();  
        myHT.Add( 0, "zero" );  
        myHT.Add( 1, "one" );  
        myHT.Add( 2, "two" );  
        myHT.Add( 3, "three" );  
        myHT.Add( 4, "four" );  
  
        // Creates a synchronized wrapper around the Hashtable.  
        Hashtable mySyncdHT = Hashtable.Synchronized( myHT );  
  
        // Displays the synchronization status of both Hashtables.  
        Console.WriteLine( "myHT is {0}.", myHT.IsSynchronized ? "synchronized" : "not synchronized" );  
        Console.WriteLine( "mySyncdHT is {0}.", mySyncdHT.IsSynchronized ? "synchronized" : "not synchronized" );  
    }  
}  
/*  
This code produces the following output.  
  
myHT is not synchronized.  
mySyncdHT is synchronized.  
*/
```

## Remarks

A [Hashtable](#) can support one writer and multiple readers concurrently. To support multiple writers, all operations must be done through the wrapper returned by the [Synchronized](#) method.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration:

```
Hashtable myCollection = new Hashtable();
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

See

[Synchronized\(Hashtable\)](#)

Also

[SyncRoot](#)

[SyncRoot](#)

# Hashtable.Item[Object] Hashtable.Item[Object]

## In this Article

Gets or sets the value associated with the specified key.

```
public virtual object this[object key] { get; set; }  
member this.Item(obj) : obj with get, set
```

## Parameters

key Object Object

The key whose value to get or set.

## Returns

[Object Object](#)

The value associated with the specified key. If the specified key is not found, attempting to get it returns `null`, and attempting to set it creates a new element using the specified key.

## Exceptions

[ArgumentNullException ArgumentNullException](#)

`key` is `null`.

[NotSupportedException NotSupportedException](#)

The property is set and the [Hashtable](#) is read-only.

-or-

The property is set, `key` does not exist in the collection, and the [Hashtable](#) has a fixed size.

## Remarks

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\[Object\]](#) property to add new elements by setting the value of a key that does not exist in the [Hashtable](#); for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the [Hashtable](#), setting the [Item\[Object\]](#) 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. To distinguish between `null` that is returned because the specified key is not found and `null` that is returned because the value of the specified key is `null`, use the [Contains](#) method or the [ContainsKey](#) method to determine if the key exists in the list.

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

The C# language uses the keyword to define the indexers instead of implementing the [Item\[Int32\]](#) property. Visual Basic implements [Item\[Object\]](#) as a default property, which provides the same indexing functionality.

See

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

Also

# Hashtable.KeyEquals Hashtable.KeyEquals

## In this Article

Compares a specific [Object](#) with a specific key in the [Hashtable](#).

```
protected virtual bool KeyEquals (object item, object key);  
abstract member KeyEquals : obj * obj -> bool  
override this.KeyEquals : obj * obj -> bool
```

## Parameters

item Object Object

The [Object](#) to compare with `key`.

key Object Object

The key in the [Hashtable](#) to compare with `item`.

## Returns

[Boolean](#) Boolean

`true` if `item` and `key` are equal; otherwise, `false`.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`item` is `null`.

-or-

`key` is `null`.

## Remarks

If the hash table was created with a specific [IComparer](#) implementation, this method uses that comparer; that is, `Compare(item, key)`. Otherwise, it uses `item.Equals(key)`.

This method is an O(1) operation.

## See

Also

[Object](#)[Object](#)

[Equals\(Object\)](#)[Equals\(Object\)](#)

# Hashtable.Keys Hashtable.Keys

## In this Article

Gets an [ICollection](#) containing the keys in the [Hashtable](#).

```
public virtual System.Collections.ICollection Keys { get; }
```

```
member this.Keys : System.Collections.ICollection
```

Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) containing the keys in the [Hashtable](#).

## Remarks

The order of the keys in the [ICollection](#) is unspecified, but it is the same order as the associated values in the [ICollection](#) returned by the [Values](#) method.

The returned [ICollection](#) is not a static copy; instead, the [ICollection](#) refers back to the keys in the original [Hashtable](#). Therefore, changes to the [Hashtable](#) continue to be reflected in the [ICollection](#).

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

See

[ICollection](#)  
[Collection](#)

Also

[Values](#)  
[Values](#)

# Hashtable.OnDeserialization Hashtable.OnDeserialization

## In this Article

Implements the [ISerializable](#) interface and raises the deserialization event when the deserialization is complete.

```
public virtual void OnDeserialization (object sender);  
  
abstract member OnDeserialization : obj -> unit  
override this.OnDeserialization : obj -> unit
```

## Parameters

sender	<a href="#">Object</a> <a href="#">Object</a>
--------	---

The source of the deserialization event.

## Exceptions

[SerializationException](#) [SerializationException](#)

The [SerializationInfo](#) object associated with the current [Hashtable](#) is invalid.

## Remarks

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

Because serializing and deserializing an enumerator for a [Hashtable](#) can cause the elements to become reordered, it is not possible to continue enumeration without calling the [Reset](#) method.

See [ISerializable](#)  
[Serializable](#)

Also [GetObjectData\(SerializationInfo, StreamingContext\)](#)  
[GetObjectData\(SerializationInfo, StreamingContext\)](#)

# Hashtable.Remove Hashtable.Remove

## In this Article

Removes the element with the specified key from the [Hashtable](#).

```
public virtual void Remove (object key);  
  
abstract member Remove : obj -> unit  
override this.Remove : obj -> unit
```

## Parameters

key	<a href="#">Object</a>
-----	------------------------

The key of the element to remove.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`key` is `null`.

[NotSupportedException](#) [NotSupportedException](#)

The [Hashtable](#) is read-only.

-or-

The [Hashtable](#) has a fixed size.

## Examples

The following example shows how to remove elements from the [Hashtable](#).

```
using System;  
using System.Collections;  
public class SamplesHashtable {  
  
    public static void Main() {  
  
        // Creates and initializes a new Hashtable.  
        Hashtable myHT = new Hashtable();  
        myHT.Add( "1a", "The" );  
        myHT.Add( "1b", "quick" );  
        myHT.Add( "1c", "brown" );  
        myHT.Add( "2a", "fox" );  
        myHT.Add( "2b", "jumps" );  
        myHT.Add( "2c", "over" );  
        myHT.Add( "3a", "the" );  
        myHT.Add( "3b", "lazy" );  
        myHT.Add( "3c", "dog" );  
  
        // Displays the Hashtable.  
        Console.WriteLine( "The Hashtable initially contains the following:" );  
        PrintKeysAndValues( myHT );  
  
        // Removes the element with the key "3b".  
        myHT.Remove( "3b" );  
  
        // Displays the current state of the Hashtable.  
        Console.WriteLine( "After removing \"lazy\": " );  
        PrintKeysAndValues( myHT );  
    }  
}
```

```

public static void PrintKeysAndValues( Hashtable myHT ) {
    foreach ( DictionaryEntry de in myHT )
        Console.WriteLine( "    {0}:    {1}", de.Key, de.Value );
    Console.WriteLine();
}

/*
This code produces the following output.

The Hashtable initially contains the following:
2c:    over
3a:    the
2b:    jumps
3b:    lazy
1b:    quick
3c:    dog
2a:    fox
1c:    brown
1a:    The

After removing "lazy":
2c:    over
3a:    the
2b:    jumps
1b:    quick
3c:    dog
2a:    fox
1c:    brown
1a:    The
*/

```

## Remarks

If the [Hashtable](#) does not contain an element with the specified key, the [Hashtable](#) remains unchanged. No exception is thrown.

This method is an O(1) operation.

See

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

Also

# Hashtable.Synchronized Hashtable.Synchronized

## In this Article

Returns a synchronized (thread-safe) wrapper for the [Hashtable](#).

```
public static System.Collections.Hashtable Synchronized (System.Collections.Hashtable table);  
static member Synchronized : System.Collections.Hashtable -> System.Collections.Hashtable
```

## Parameters

table [Hashtable](#) [Hashtable](#)

The [Hashtable](#) to synchronize.

## Returns

[Hashtable](#) [Hashtable](#)

A synchronized (thread-safe) wrapper for the [Hashtable](#).

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

table is null.

## Examples

The following example shows how to synchronize a [Hashtable](#), determine if a [Hashtable](#) is synchronized, and use a synchronized [Hashtable](#).

```
using System;  
using System.Collections;  
public class SamplesHashtable {  
  
    public static void Main() {  
  
        // Creates and initializes a new Hashtable.  
        Hashtable myHT = new Hashtable();  
        myHT.Add( 0, "zero" );  
        myHT.Add( 1, "one" );  
        myHT.Add( 2, "two" );  
        myHT.Add( 3, "three" );  
        myHT.Add( 4, "four" );  
  
        // Creates a synchronized wrapper around the Hashtable.  
        Hashtable mySyncdHT = Hashtable.Synchronized( myHT );  
  
        // Displays the synchronization status of both Hashtables.  
        Console.WriteLine( "myHT is {0}.", myHT.IsSynchronized ? "synchronized" : "not synchronized" );  
        Console.WriteLine( "mySyncdHT is {0}.", mySyncdHT.IsSynchronized ? "synchronized" : "not synchronized" );  
    }  
}  
/*  
This code produces the following output.  
  
myHT is not synchronized.  
mySyncdHT is synchronized.  
*/
```

## Remarks

The [Synchronized](#) method is thread safe for multiple readers and writers. Furthermore, the synchronized wrapper ensures that there is only one writer writing at a time.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration:

```
Hashtable myCollection = new Hashtable();
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

This method is an O(1) operation.

See

[IsSynchronized](#)

Also

[SyncRoot](#)

# Hashtable.SyncRoot Hashtable.SyncRoot

## In this Article

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

```
public virtual object SyncRoot { get; }  
member this.SyncRoot : obj
```

Returns

[Object Object](#)

An object that can be used to synchronize access to the [Hashtable](#).

## Remarks

To create a synchronized version of the [Hashtable](#), use the [Synchronized](#) method. However, derived classes can provide their own synchronized version of the [Hashtable](#) using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) of the [Hashtable](#), not directly on the [Hashtable](#). This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [Hashtable](#) object.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration:

```
Hashtable myCollection = new Hashtable();  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

See

[IsSynchronized](#)  
[IsSynchronized](#)

Also

[Synchronized\(Hashtable\)](#)  
[Synchronized\(Hashtable\)](#)

# Hashtable.Values Hashtable.Values

## In this Article

Gets an [ICollection](#) containing the values in the [Hashtable](#).

```
public virtual System.Collections.ICollection Values { get; }  
member this.Values : System.Collections.ICollection
```

Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) containing the values in the [Hashtable](#).

## Remarks

The order of the values in the [ICollection](#) is unspecified, but it is the same order as the associated keys in the [ICollection](#) returned by the [Keys](#) method.

The returned [ICollection](#) is not a static copy; instead, the [ICollection](#) refers back to the values in the original [Hashtable](#). Therefore, changes to the [Hashtable](#) continue to be reflected in the [ICollection](#).

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

See

[ICollection](#)  
[Collection](#)

Also

[Keys](#)  
[Keys](#)

# **ICollection** ICollection Interface

Defines size, enumerators, and synchronization methods for all nongeneric collections.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface ICollection : System.Collections.IEnumerable

type ICollection = interface
    interface IEnumerable
```

## Inheritance Hierarchy

None

## Remarks

The [ICollection](#) interface is the base interface for classes in the [System.Collections](#) namespace. Its generic equivalent is the [System.Collections.Generic.ICollection<T>](#) interface.

The [ICollection](#) interface extends [IEnumerable](#); [IDictionary](#) and [IList](#) are more specialized interfaces that extend [ICollection](#). An [IDictionary](#) implementation is a collection of key/value pairs, like the [Hashtable](#) class. An [IList](#) implementation is a collection of values and its members can be accessed by index, like the [ArrayList](#) class.

Some collections that limit access to their elements, such as the [Queue](#) class and the [Stack](#) class, directly implement the [ICollection](#) interface.

If neither the [IDictionary](#) interface nor the [IList](#) interface meet the requirements of the required collection, derive the new collection class from the [ICollection](#) interface instead for more flexibility.

For the generic version of this interface, see [System.Collections.Generic.ICollection<T>](#).

## Properties

Count

Count

Gets the number of elements contained in the [ICollection](#).

IsSynchronized

IsSynchronized

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

SyncRoot

SyncRoot

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

## Methods

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

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

# ICollection.CopyTo ICollection.CopyTo

## In this Article

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

```
public void CopyTo (Array array, int index);  
abstract member CopyTo : Array * int -> unit
```

## Parameters

array [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](#) [Int32](#)

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

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#) [ArgumentException](#)

[array](#) is multidimensional.

-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](#).

# **ICollection.Count**

## In this Article

Gets the number of elements contained in the [ICollection](#).

```
public int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [ICollection](#).

# **ICollection.IsSynchronized**

## In this Article

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

```
public bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`.

## Remarks

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

Most collection classes in the [System.Collections](#) namespace also implement a [Synchronized](#) method, which provides a synchronized wrapper around the underlying collection.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) property during the entire enumeration.

```
ICollection myCollection = someCollection;  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

See

[SyncRoot](#)

Also

# ICollection.SyncRoot ICollection.SyncRoot

## In this Article

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

```
public object SyncRoot { get; }  
member this.SyncRoot : obj
```

Returns

[Object Object](#)

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

## Remarks

For collections whose underlying store is not publicly available, the expected implementation is to return the current instance. Note that the pointer to the current instance might not be sufficient for collections that wrap other collections; those should return the underlying collection's `SyncRoot` property.

Most collection classes in the [System.Collections](#) namespace also implement a `Synchronized` method, which provides a synchronized wrapper around the underlying collection. However, derived classes can provide their own synchronized version of the collection using the `SyncRoot` property. The synchronizing code must perform operations on the `SyncRoot` property of the collection, not directly on the collection. This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the collection instance.

In the absence of a `Synchronized` method on a collection, the expected usage for `SyncRoot` looks as follows:

```
ICollection myCollection = someCollection;  
lock(myCollection.SyncRoot)  
{  
    // Some operation on the collection, which is now thread safe.  
}
```

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 causes 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 following code example shows how to lock the collection using the `SyncRoot` property during the entire enumeration.

```
ICollection myCollection = someCollection;  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

See

[IsSynchronized](#)[IsSynchronized](#)

Also

# IComparer IComparer Interface

Exposes a method that compares two objects.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface IComparer

type IComparer = interface
```

## Inheritance Hierarchy

None

## Remarks

This interface is used in conjunction with the [Array.Sort](#) and [Array.BinarySearch](#) methods. It provides a way to customize the sort order of a collection. See the [Compare](#) method for notes on parameters and return value. Its generic equivalent is the [System.Collections.Generic.IComparer<T>](#) interface.

The default implementation of this interface is the [Comparer](#) class. For the generic version of this interface, see [System.Collections.Generic.IComparer<T>](#).

## Methods

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

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

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

# IComparer.Compare

## In this Article

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

```
public int Compare (object x, object y);  
abstract member Compare : obj * obj -> int
```

### Parameters

x Object Object

The first object to compare.

y Object Object

The second object to compare.

### Returns

Int32 Int32

A signed integer that indicates the relative values of `x` and `y`:

- If less than 0, `x` is less than `y`.
- If 0, `x` equals `y`.
- If greater than 0, `x` is greater than `y`.

### Exceptions

[ArgumentException](#) [ArgumentException](#)

Neither `x` nor `y` implements the [IComparable](#) interface.

-or-

`x` and `y` are of different types and neither one can handle comparisons with the other.

## Examples

The following example uses the [IComparer](#) interface to sort a string array. In this example, the `Compare` method is implemented using the [CaseInsensitiveComparer](#) class to reverse the order of the contents of the array.

```
using System;  
using System.Collections;  
  
public class Example  
{  
    public class ReverserClass : IComparer  
    {  
        // Call CaseInsensitiveComparer.Compare with the parameters reversed.  
        int IComparer.Compare(Object x, Object y)  
        {  
            return ((new CaseInsensitiveComparer()).Compare(y, x));  
        }  
    }  
  
    public static void Main()  
    {  
        // Initialize a string array.  
        string[] words = { "The", "quick", "brown", "fox", "jumps", "over",  
                          "the", "lazy", "dog" };  
    }  
}
```

```

// Display the array values.
Console.WriteLine("The array initially contains the following values:" );
PrintIndexAndValues(words);

// Sort the array values using the default comparer.
Array.Sort(words);
Console.WriteLine("After sorting with the default comparer:" );
PrintIndexAndValues(words);

// Sort the array values using the reverse case-insensitive comparer.
Array.Sort(words, new ReverserClass());
Console.WriteLine("After sorting with the reverse case-insensitive comparer:" );
PrintIndexAndValues(words);

}

public static void PrintIndexAndValues(IEnumerable list)
{
    int i = 0;
    foreach (var item in list )
        Console.WriteLine($"    [{i++}]: {item}");

    Console.WriteLine();
}

}

// The example displays the following output:
//      The array initially contains the following values:
//      [0]: The
//      [1]: quick
//      [2]: brown
//      [3]: fox
//      [4]: jumps
//      [5]: over
//      [6]: the
//      [7]: lazy
//      [8]: dog
//
//      After sorting with the default comparer:
//      [0]: brown
//      [1]: dog
//      [2]: fox
//      [3]: jumps
//      [4]: lazy
//      [5]: over
//      [6]: quick
//      [7]: the
//      [8]: The
//
//      After sorting with the reverse case-insensitive comparer:
//      [0]: the
//      [1]: The
//      [2]: quick
//      [3]: over
//      [4]: lazy
//      [5]: jumps
//      [6]: fox
//      [7]: dog
//      [8]: brown

```

## Remarks

The preferred implementation is to use the [CompareTo](#) method of one of the parameters.

Comparing `null` with any type is allowed and does not generate an exception when using [IComparable](#). When sorting, `null` is considered to be less than any other object.

# IDictionary

## IDictionary Interface

Represents a nongeneric collection of key/value pairs.

### Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface IDictionary : System.Collections.ICollection

type IDictionary = interface
    interface ICollection
    interface IEnumerable
```

### Inheritance Hierarchy

None

## Remarks

The [IDictionary](#) interface is the base interface for nongeneric collections of key/value pairs. For the generic version of this interface, see [System.Collections.Generic.IDictionary< TKey, TValue >](#).

Each element is a key/value pair stored in a [DictionaryEntry](#) object.

Each pair must have a unique key. Implementations can vary in whether they allow the key to be null. The value can be null and does not have to be unique. The [IDictionary](#) interface allows the contained keys and values to be enumerated, but it does not imply any particular sort order.

[IDictionary](#) implementations fall into three categories: read-only, fixed-size, variable-size. A read-only [IDictionary](#) object cannot be modified. A fixed-size [IDictionary](#) object does not allow the addition or removal of elements, but does allow the modification of existing elements. A variable-size [IDictionary](#) object allows the addition, removal, and modification of elements.

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](#) object is a key/value pair, the element type is not the type of the key or the type of the value. Instead, the element type is [DictionaryEntry](#). For example:

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

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

## Properties

[IsFixedSize](#)

[IsFixedSize](#)

Gets a value indicating whether the [IDictionary](#) object has a fixed size.

[IsReadOnly](#)

[IsReadOnly](#)

Gets a value indicating whether the [IDictionary](#) object is read-only.

`Item[Object]`

`Item[Object]`

Gets or sets the element with the specified key.

`Keys`

`Keys`

Gets an `ICollection` object containing the keys of the `IDictionary` object.

`Values`

`Values`

Gets an `ICollection` object containing the values in the `IDictionary` object.

## Methods

`Add(Object, Object)`

`Add(Object, Object)`

Adds an element with the provided key and value to the `IDictionary` object.

`Clear()`

`Clear()`

Removes all elements from the `IDictionary` object.

`Contains(Object)`

`Contains(Object)`

Determines whether the `IDictionary` object contains an element with the specified key.

`GetEnumerator()`

`GetEnumerator()`

Returns an `IDictionaryEnumerator` object for the `IDictionary` object.

`Remove(Object)`

`Remove(Object)`

Removes the element with the specified key from the `IDictionary` object.

## See Also

[ICollection](#) [ICollection](#)

# IDictionary.Add

## In this Article

Adds an element with the provided key and value to the [IDictionary](#) object.

```
public void Add (object key, object value);  
abstract member Add : obj * obj -> unit
```

## Parameters

key [Object](#) [Object](#)

The [Object](#) to use as the key of the element to add.

value [Object](#) [Object](#)

The [Object](#) to use as the value of the element to add.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#) [ArgumentException](#)

An element with the same key already exists in the [IDictionary](#) object.

[NotSupportedException](#) [NotSupportedException](#)

The [IDictionary](#) is read-only.

-or-

The [IDictionary](#) has a fixed size.

## Examples

The following code example demonstrates how to implement the `Add` method. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public void Add(object key, object value)  
{  
    // Add the new key/value pair even if this key already exists in the dictionary.  
    if (ItemsInUse == items.Length)  
        throw new InvalidOperationException("The dictionary cannot hold any more items.");  
    items[ItemsInUse++] = new DictionaryEntry(key, value);  
}
```

## Remarks

You can also use the [Item\[Object\]](#) 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\[Object\]](#) property overwrites the old value. In contrast, the `Add` method does not modify existing elements.

Implementations can vary in whether they allow the key to be `null`.

See

[Item\[Object\]](#)[Item\[Object\]](#)

Also

# IDictionary.Clear IDictionary.Clear

## In this Article

Removes all elements from the [IDictionary](#) object.

```
public void Clear ();  
abstract member Clear : unit -> unit
```

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [IDictionary](#) object is read-only.

## Examples

The following code example demonstrates how to implement the [Clear](#) method. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public void Clear() { ItemsInUse = 0; }
```

# IDictionary.Contains IDictionary.Contains

## In this Article

Determines whether the [IDictionary](#) object contains an element with the specified key.

```
public bool Contains (object key);  
abstract member Contains : obj -> bool
```

### Parameters

key Object Object

The key to locate in the [IDictionary](#) object.

### Returns

[Boolean](#) Boolean

`true` if the [IDictionary](#) contains an element with the key; otherwise, `false`.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

## Examples

The following code example demonstrates how to implement the [Contains](#) method. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public bool Contains(object key)  
{  
    Int32 index;  
    return TryGetIndexByKey(key, out index);  
}
```

## Remarks

Implementations can vary in whether they allow the key to be `null`.

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether `item` exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

# IDictionary.GetEnumerator IDictionary.GetEnumerator

## In this Article

Returns an [IDictionaryEnumerator](#) object for the [IDictionary](#) object.

```
public System.Collections.IDictionaryEnumerator GetEnumerator ();  
abstract member GetEnumerator : unit -> System.Collections.IDictionaryEnumerator
```

Returns

[IDictionaryEnumerator](#) [IDictionaryEnumerator](#)

An [IDictionaryEnumerator](#) object for the [IDictionary](#) object.

## Remarks

[Visual Basic, C#]

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 [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 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.

See

[IDictionaryEnumerator](#)[IDictionaryEnumerator](#)

Also

# IDictionary.IsFixedSize IDictionary.IsFixedSize

## In this Article

Gets a value indicating whether the [IDictionary](#) object has a fixed size.

```
public bool IsFixedSize { get; }  
member this.IsFixedSize : bool
```

Returns

[Boolean](#) Boolean

`true` if the [IDictionary](#) object has a fixed size; otherwise, `false`.

## Examples

The following code example demonstrates how to implement the [IsFixedSize](#) property. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public bool IsFixedSize { get { return false; } }
```

## Remarks

A collection with a fixed size does not allow the addition or removal of elements after the collection is created, but does allow 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.

# IDictionary.IsReadOnly IDictionary.IsReadOnly

## In this Article

Gets a value indicating whether the [IDictionary](#) object is read-only.

```
public bool IsReadOnly { get; }  
member this.IsReadOnly : bool
```

Returns

[Boolean](#)

`true` if the [IDictionary](#) object is read-only; otherwise, `false`.

## Examples

The following code example demonstrates how to implement the [IsReadOnly](#) property. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public bool IsReadOnly { get { return false; } }
```

## Remarks

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

# IDictionary.Item[Object] IDictionary.Item[Object]

## In this Article

Gets or sets the element with the specified key.

```
public object this[object key] { get; set; }  
member this.Item(obj) : obj with get, set
```

### Parameters

key Object Object

The key of the element to get or set.

### Returns

[Object Object](#)

The element with the specified key, or `null` if the key does not exist.

### Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`key` is `null`.

[NotSupportedException](#) [NotSupportedException](#)

The property is set and the [IDictionary](#) object is read-only.

-or-

The property is set, `key` does not exist in the collection, and the [IDictionary](#) has a fixed size.

## Examples

The following code example demonstrates how to implement the [Item\[Object\]](#) property. This code example is part of a larger example provided for the [IDictionary](#) class.

```

public object this[object key]
{
    get
    {
        // If this key is in the dictionary, return its value.
        Int32 index;
        if (TryGetIndexOfKey(key, out index))
        {
            // The key was found; return its value.
            return items[index].Value;
        }
        else
        {
            // The key was not found; return null.
            return null;
        }
    }

    set
    {
        // If this key is in the dictionary, change its value.
        Int32 index;
        if (TryGetIndexOfKey(key, out index))
        {
            // The key was found; change its value.
            items[index].Value = value;
        }
        else
        {
            // This key is not in the dictionary; add this key/value pair.
            Add(key, value);
        }
    }
}

```

## Remarks

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\[Object\]](#) 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\[Object\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

Implementations can vary in whether they allow the key to be `null`.

The C# language uses the `this` keyword to define the indexers instead of implementing the [Item\[Object\]](#) property. Visual Basic implements [Item\[Object\]](#) as a default property, which provides the same indexing functionality.

See

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

Also

# IDictionary.Keys IDictionary.Keys

## In this Article

Gets an [ICollection](#) object containing the keys of the [IDictionary](#) object.

```
public System.Collections.ICollection Keys { get; }  
member this.Keys : System.Collections.ICollection
```

Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) object containing the keys of the [IDictionary](#) object.

## Examples

The following code example demonstrates how to implement the [Keys](#) property. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public ICollection Keys  
{  
    get  
    {  
        // Return an array where each item is a key.  
        Object[] keys = new Object[ItemsInUse];  
        for (Int32 n = 0; n < ItemsInUse; n++)  
            keys[n] = items[n].Key;  
        return keys;  
    }  
}
```

## Remarks

The order of the keys in the returned [ICollection](#) object is unspecified, but is guaranteed to be the same order as the corresponding values in the [ICollection](#) returned by the [Values](#) property.

See

[ICollection](#) [Collection](#)

Also

# IDictionary.Remove

## In this Article

Removes the element with the specified key from the [IDictionary](#) object.

```
public void Remove (object key);  
abstract member Remove : obj -> unit
```

### Parameters

key	<a href="#">Object</a>
-----	------------------------

The key of the element to remove.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

[NotSupportedException](#) [NotSupportedException](#)

The [IDictionary](#) object is read-only.

-or-

The [IDictionary](#) has a fixed size.

## Examples

The following code example demonstrates how to implement the [Remove](#) method. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public void Remove(object key)  
{  
    if (key == null) throw new ArgumentNullException("key");  
    // Try to find the key in the DictionaryEntry array  
    Int32 index;  
    if (TryGetIndexOfKey(key, out index))  
    {  
        // If the key is found, slide all the items up.  
        Array.Copy(items, index + 1, items, index, ItemsInUse - index - 1);  
        ItemsInUse--;  
    }  
    else  
    {  
        // If the key is not in the dictionary, just return.  
    }  
}
```

## Remarks

If the [IDictionary](#) object does not contain an element with the specified key, the [IDictionary](#) remains unchanged. No exception is thrown.

# IDictionary.Values IDictionary.Values

## In this Article

Gets an [ICollection](#) object containing the values in the [IDictionary](#) object.

```
public System.Collections.ICollection Values { get; }  
member this.Values : System.Collections.ICollection
```

Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) object containing the values in the [IDictionary](#) object.

## Examples

The following code example demonstrates how to implement the [Values](#) property. This code example is part of a larger example provided for the [IDictionary](#) class.

```
public ICollection Values  
{  
    get  
    {  
        // Return an array where each item is a value.  
        Object[] values = new Object[ItemsInUse];  
        for (Int32 n = 0; n < ItemsInUse; n++)  
            values[n] = items[n].Value;  
        return values;  
    }  
}
```

## Remarks

The order of the values in the returned [ICollection](#) object is unspecified, but is guaranteed to be the same order as the corresponding keys in the [ICollection](#) returned by the [Keys](#) property.

See

[ICollection](#)[ICollection](#)

Also

# IDictionaryEnumerator IDictionaryEnumerator Interface

Enumerates the elements of a nongeneric dictionary.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface IDictionaryEnumerator : System.Collections.IEnumerator

type IDictionaryEnumerator = interface
    interface IEnumerator
```

## Inheritance Hierarchy

None

## 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. The `Reset` method also brings the enumerator back to this position. 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 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. Even when a collection is synchronized, other threads can still modify the collection, which causes 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.

## Properties

Entry

Entry

Gets both the key and the value of the current dictionary entry.

Key

Key

Gets the key of the current dictionary entry.

**Value**

**Value**

Gets the value of the current dictionary entry.

## See Also

[IEnumerator](#) [IEnumerator](#)

# IDictionaryEnumerator.Entry IDictionaryEnumerator.Entry

## In this Article

Gets both the key and the value of the current dictionary entry.

```
public System.Collections.DictionaryEntry Entry { get; }  
member this.Entry : System.Collections.DictionaryEntry
```

Returns

[DictionaryEntry](#) [DictionaryEntry](#)

A [DictionaryEntry](#) containing both the key and the value of the current dictionary entry.

## Remarks

[Entry](#) 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 [Entry](#).
- 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.

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

See

[Key](#)

Also

[Value](#)

# IDictionaryEnumerator.Key IDictionaryEnumerator.Key

## In this Article

Gets the key of the current dictionary entry.

```
public object Key { get; }  
member this.Key : obj
```

Returns

[Object Object](#)

The key of the current element of the enumeration.

## Remarks

[Key](#) 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 [Key](#).
- 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.

[Key](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Key](#) to the key of the next element in enumeration.

See

[Value Value](#)

Also

[Entry Entry](#)

# IDictionaryEnumerator.Value IDictionaryEnumerator.Value

## Value

### In this Article

Gets the value of the current dictionary entry.

```
public object Value { get; }  
member this.Value : obj
```

### Returns

[Object Object](#)

The value of the current element of the enumeration.

## Remarks

[Value](#) 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 [Value](#).
- 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.

[Value](#) returns the same object until [MoveNext](#) is called. [MoveNext](#) sets [Value](#) to the value of the next element in enumeration.

See

[Key Key](#)

Also

[Entry Entry](#)

# IEnumerable Interface

Exposes an enumerator, which supports a simple iteration over a non-generic collection.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[System.Runtime.InteropServices.Guid("496B0ABE-CDEE-11d3-88E8-00902754C43A")]
public interface IEnumerable

type IEnumerable = interface
```

## Inheritance Hierarchy

None

## Remarks

[IEnumerable](#) is the base interface for all non-generic collections that can be enumerated. For the generic version of this interface see [System.Collections.Generic.IEnumerable<T>](#). [IEnumerable](#) contains a single method, [GetEnumerator](#), which returns an [IEnumerator](#). [IEnumerator](#) provides the ability to iterate through the collection by exposing a [Current](#) property and [MoveNext](#) and [Reset](#) methods.

It is a best practice to implement [IEnumerable](#) and [IEnumerator](#) on your collection classes to enable the `foreach` ([For Each](#) in Visual Basic) syntax, however implementing [IEnumerable](#) is not required. If your collection does not implement [IEnumerable](#), you must still follow the iterator pattern to support this syntax by providing a [GetEnumerator](#) method that returns an interface, class or struct. When using Visual Basic, you must provide an [IEnumerator](#) implementation, which is returned by [GetEnumerator](#). When developing with C# you must provide a class that contains a [Current](#) property, and [MoveNext](#) and [Reset](#) methods as described by [IEnumerator](#), but the class does not have to implement [IEnumerator](#).

## Methods

[GetEnumerator\(\)](#)

[Getenumerator\(\)](#)

Returns an enumerator that iterates through a collection.

## See Also

[IEnumerator](#) [IEnumerator](#)

[IEnumerator](#) [IEnumerator](#)

[IEnumerator](#) [IEnumerator](#)

# IEnumerable.GetEnumerator() | IEnumerable.GetEnumerator

## In this Article

Returns an enumerator that iterates through a collection.

```
public System.Collections.IEnumerator GetEnumerator ();  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

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

## Examples

The following code example demonstrates the implementation of the [IEnumerable](#) interfaces for a custom collection. In this example, [GetEnumerator](#) is not explicitly called, but it is implemented to support the use of [foreach](#) ([For Each](#) in Visual Basic). This code example is part of a larger example for the [IEnumerable](#) interface.

```
using System;  
using System.Collections;  
  
// Simple business object.  
public class Person  
{  
    public Person(string fName, string lName)  
    {  
        this.firstName = fName;  
        this.lastName = lName;  
    }  
  
    public string firstName;  
    public string lastName;  
}  
  
// Collection of Person objects. This class  
// implements IEnumerable so that it can be used  
// with ForEach syntax.  
public class People : IEnumerable  
{  
    private Person[] _people;  
    public People(Person[] pArray)  
    {  
        _people = new Person[pArray.Length];  
  
        for (int i = 0; i < pArray.Length; i++)  
        {  
            _people[i] = pArray[i];  
        }  
    }  
  
    // Implementation for the GetEnumerator method.  
    IEnumerator IEnumerable.GetEnumerator()  
    {  
        return (IEnumerator) GetEnumerator();  
    }  
  
    public PeopleEnum GetEnumerator()  
    {
```

```

        {
            return new PeopleEnum(_people);
        }
    }

// When you implement IEnumerable, you must also implement IEnumerator.
public class PeopleEnum : IEnumerator
{
    public Person[] _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(Person[] list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Length);
    }

    public void Reset()
    {
        position = -1;
    }

    object IEnumerator.Current
    {
        get
        {
            return Current;
        }
    }

    public Person Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}

class App
{
    static void Main()
    {
        Person[] peopleArray = new Person[3]
        {
            new Person("John", "Smith"),
            new Person("Jim", "Johnson"),
            new Person("Sue", "Rabon"),
        };

        People peopleList = new People(peopleArray);
    }
}

```

```
    foreach (Person p in peopleList)
        Console.WriteLine(p.firstName + " " + p.lastName);

    }

/* This code produces output similar to the following:
 *
 * John Smith
 * Jim Johnson
 * Sue Rabon
 *
 */


```

## 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. 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`.

`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` returns `false`, `Current` is undefined. To set `Current` to the first element of the collection again, you can call `Reset` followed by `MoveNext`.

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.

See

[IEnumerator](#)

Also

[Iterators \(C# and Visual Basic\)](#)

# IEnumerator IEnumerator Interface

Supports a simple iteration over a non-generic collection.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[System.Runtime.InteropServices.Guid("496B0ABF-CDEE-11d3-88E8-00902754C43A")]
public interface IEnumerator

type IEnumerator = interface
```

## Inheritance Hierarchy

None

## Remarks

[IEnumerator](#) is the base interface for all non-generic enumerators. Its generic equivalent is the [System.Collections.Generic.IEnumerator<T>](#) interface.

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 `Reset` method is provided for COM interoperability and does not need to be fully implemented; instead, the implementer can throw a [NotSupportedException](#).

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

`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`, if it's implemented, followed by `MoveNext`. If `Reset` is not implemented, you must create a new enumerator instance to return to the first element of the collection.

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. Even when a collection is synchronized, other threads can still modify the collection, which causes 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.

## Properties

`Current`

`Current`

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

## Methods

`MoveNext()`

`MoveNext()`

Advances the enumerator to the next element of the collection.

`Reset()`

`Reset()`

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

# IEnumerator.Current | IEnumerator.Current

## In this Article

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

```
public object Current { get; }  
member this.Current : obj
```

Returns

[Object](#) [Object](#)

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

## Examples

The following code example demonstrates the implementation of the [IEnumerator](#) interfaces for a custom collection. In this example, [Current](#) is not explicitly called, but it is implemented to support the use of `foreach` (`for each` in Visual Basic). This code example is part of a larger example for the [IEnumerator](#) interface.

```

// When you implement IEnumerable, you must also implement IEnumerator.
public class PeopleEnum : IEnumerator
{
    public Person[] _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(Person[] list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Length);
    }

    public void Reset()
    {
        position = -1;
    }

    object IEnumerator.Current
    {
        get
        {
            return Current;
        }
    }

    public Person Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}

```

## 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.

See

Also

[MoveNext\(\)](#)

[Reset\(\)](#)

# IEnumerator.MoveNext Ienumerator.MoveNext

## In this Article

Advances the enumerator to the next element of the collection.

```
public bool MoveNext ();  
abstract member MoveNext : unit -> bool
```

Returns

[Boolean Boolean](#)

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

## Examples

The following code example demonstrates the implementation of the [IEnumerator](#) interfaces for a custom collection. In this example, [MoveNext](#) is not explicitly called, but it is implemented to support the use of `foreach` (`for each` in Visual Basic). This code example is part of a larger example for the [IEnumerator](#) interface.

```

// When you implement IEnumerable, you must also implement IEnumerator.
public class PeopleEnum : IEnumerator
{
    public Person[] _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(Person[] list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Length);
    }

    public void Reset()
    {
        position = -1;
    }

    object IEnumerator.Current
    {
        get
        {
            return Current;
        }
    }

    public Person Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}

```

## Remarks

After an enumerator is created or after the [Reset](#) method is called, an enumerator is positioned before the first element of the collection, and the first call to the [MoveNext](#) method moves the enumerator over 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` until [Reset](#) is called.

If changes are made to the collection, such as adding, modifying, or deleting elements, the behavior of [MoveNext](#) is undefined.

See

Also

[Current](#)

[Reset\(\)](#)

# IEnumerator.Reset Ienumerator.Reset

## In this Article

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

```
public void Reset ();  
abstract member Reset : unit -> unit
```

## Examples

The following code example demonstrates the implementation of the [IEnumerator](#) interfaces for a custom collection. In this example, [Reset](#) is not explicitly called, but it is implemented to support the use of [foreach](#) ([for each](#) in Visual Basic). This code example is part of a larger example for the [IEnumerator](#) interface.

```

// When you implement IEnumerable, you must also implement IEnumerator.
public class PeopleEnum : IEnumerator
{
    public Person[] _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(Person[] list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Length);
    }

    public void Reset()
    {
        position = -1;
    }

    object IEnumerator.Current
    {
        get
        {
            return Current;
        }
    }

    public Person Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}

```

## Remarks

If changes are made to the collection, such as adding, modifying, or deleting elements, the behavior of [Reset](#) is undefined.

The [Reset](#) method is provided for COM interoperability. It does not necessarily need to be implemented; instead, the implementer can simply throw a [NotSupportedException](#).

See

[MoveNext\(\)](#)

Also

[Current](#)

# IEqualityComparer IEqualityComparer Interface

Defines methods to support the comparison of objects for equality.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface IEqualityComparer
{
    type IEqualityComparer = interface
```

## Inheritance Hierarchy

None

## Remarks

This interface allows the implementation of customized equality comparison for collections. That is, you can create your own definition of equality, and specify that this definition be used with a collection type that accepts the [IEqualityComparer](#) interface. In the .NET Framework, constructors of the [Hashtable](#), [NameValueCollection](#), and [OrderedDictionary](#) collection types accept this interface.

For the generic version of this interface, see [System.Collections.Generic.IEqualityComparer<T>](#).

The [IEqualityComparer](#) interface supports only equality comparisons. Customization of comparisons for sorting and ordering is provided by the [IComparer](#) interface.

## Methods

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

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

Determines whether the specified objects are equal.

[GetHashCode\(Object\)](#)

[GetHashCode\(Object\)](#)

Returns a hash code for the specified object.

# IEqualityComparer.Equals IEqualityComparer.Equals

## In this Article

Determines whether the specified objects are equal.

```
public bool Equals (object x, object y);  
abstract member Equals : obj * obj -> bool
```

### Parameters

x Object Object

The first object to compare.

y Object Object

The second object to compare.

### Returns

Boolean Boolean

`true` if the specified objects are equal; otherwise, `false`.

### Exceptions

ArgumentException ArgumentException

`x` and `y` are of different types and neither one can handle comparisons with the other.

## Examples

The following code example demonstrates the implementation of a case-insensitive [IEqualityComparer](#). In this example, the [CaseInsensitiveComparer.Compare](#) method is used to determine whether two objects are equal, based on the provided [CultureInfo](#).

```
class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        return obj.ToString().ToLower().GetHashCode();
    }
}
```

## Remarks

Implement this method to provide a customized equality comparison for objects.

# IEqualityComparer.GetHashCode IEqualityComparer.GetHashCode

## In this Article

Returns a hash code for the specified object.

```
public int GetHashCode (object obj);  
abstract member GetHashCode : obj -> int
```

## Parameters

obj Object Object

The [Object](#) for which a hash code is to be returned.

## Returns

[Int32 Int32](#)

A hash code for the specified object.

## Exceptions

[ArgumentNullException ArgumentNullException](#)

The type of `obj` is a reference type and `obj` is `null`.

## Examples

The following code example demonstrates the implementation of a case-insensitive [IEqualityComparer](#). In this example, the [GetHashCode](#) method returns the hash code provided by the [Object](#) type.

```
class myCultureComparer : IEqualityComparer
{
    public CaseInsensitiveComparer myComparer;

    public myCultureComparer()
    {
        myComparer = CaseInsensitiveComparer.DefaultInvariant;
    }

    public myCultureComparer(CultureInfo myCulture)
    {
        myComparer = new CaseInsensitiveComparer(myCulture);
    }

    public new bool Equals(object x, object y)
    {
        if (myComparer.Compare(x, y) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public int GetHashCode(object obj)
    {
        return obj.ToString().ToLower().GetHashCode();
    }
}
```

## Remarks

Implement this method to provide customized hash codes for objects, corresponding to the customized equality comparison provided by the [Equals](#) method.

# IHashCodeProvider IHashCodeProvider Interface

Supplies a hash code for an object, using a custom hash function.

## Declaration

```
[System.Obsolete("Please use IEqualityComparer instead.")]
[System.Runtime.InteropServices.ComVisible(true)]
public interface IHashCodeProvider

type IHashCodeProvider = interface
```

## Inheritance Hierarchy

None

## Remarks

### Important

We don't recommend that you use the `IHashCodeProvider` interface for new development. Its recommended replacement is the `System.Collections.IEqualityComparer` or `<x:Type System.Collections.Generic.IEqualityComparer>` interface.

The `IHashCodeProvider` interface is used in conjunction with the `Hashtable` class. The objects used as keys by a `Hashtable` object must override the `Object.GetHashCode` and `Object.Equals` methods. `Object.GetHashCode` or the key's implementation of `Object.GetHashCode` is used as the hash code provider. `Object.Equals` or the key's implementation of `Object.Equals` is used as the comparer.

However, some overloads of the `Hashtable` constructor take a parameter that is an `IHashCodeProvider` implementation, or a parameter that is an `IComparer` implementation, or both. If an `IHashCodeProvider` implementation is passed to the constructor, the `IHashCodeProvider.GetHashCode` method of that implementation is used as the hash code provider. If an `IComparer` implementation is passed to the constructor, the `IComparer.Compare` method of that implementation is used as the comparer.

## Methods

`GetHashCode(Object)`

`GetHashCode(Object)`

Returns a hash code for the specified object.

# IHashCodeProvider.GetHashCode IHashCodeProvider. GetHashCode

## In this Article

Returns a hash code for the specified object.

```
public int GetHashCode (object obj);  
abstract member GetHashCode : obj -> int
```

## Parameters

obj Object Object

The [Object](#) for which a hash code is to be returned.

## Returns

[Int32](#) [Int32](#)

A hash code for the specified object.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

The type of `obj` is a reference type and `obj` is `null`.

## Remarks

The return value from this method must not be persisted for two reasons. First, the hash function of a class might be altered to generate a better distribution, rendering any values from the old hash function useless. Second, the default implementation of this class does not guarantee that the same value will be returned by different instances.

# IList IList Interface

Represents a non-generic collection of objects that can be individually accessed by index.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface IList : System.Collections.ICollection

type IList = interface
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

None

## Remarks

[IList](#) is a descendant of the [ICollection](#) interface and is the base interface of all non-generic lists. [IList](#) implementations fall into three categories: read-only, fixed-size, and variable-size. A read-only [IList](#) cannot be modified. A fixed-size [IList](#) does not allow the addition or removal of elements, but it allows the modification of existing elements. A variable-size [IList](#) allows the addition, removal, and modification of elements.

For the generic version of this interface, see [System.Collections.Generic.IList<T>](#).

## Properties

[IsFixedSize](#)

[IsFixedSize](#)

Gets a value indicating whether the [IList](#) has a fixed size.

[IsReadOnly](#)

[IsReadOnly](#)

Gets a value indicating whether the [IList](#) is read-only.

[Item\[Int32\]](#)

[Item\[Int32\]](#)

Gets or sets the element at the specified index.

## Methods

[Add\(Object\)](#)

[Add\(Object\)](#)

Adds an item to the [IList](#).

[Clear\(\)](#)

[Clear\(\)](#)

Removes all items from the [IList](#).

[Contains\(Object\)](#)

[Contains\(Object\)](#)

Determines whether the [IList](#) contains a specific value.

[IndexOf\(Object\)](#)

[IndexOf\(Object\)](#)

Determines the index of a specific item in the [IList](#).

[Insert\(Int32, Object\)](#)

[Insert\(Int32, Object\)](#)

Inserts an item to the [IList](#) at the specified index.

[Remove\(Object\)](#)

[Remove\(Object\)](#)

Removes the first occurrence of a specific object from the [IList](#).

[RemoveAt\(Int32\)](#)

[RemoveAt\(Int32\)](#)

Removes the [IList](#) item at the specified index.

## See Also

[ICollection](#) [ICollection](#)

# IList.Add

## In this Article

Adds an item to the [IList](#).

```
public int Add (object value);  
abstract member Add : obj -> int
```

### Parameters

value Object Object

The object to add to the [IList](#).

### Returns

[Int32](#) [Int32](#)

The position into which the new element was inserted, or -1 to indicate that the item was not inserted into the collection.

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [IList](#) is read-only.

-or-

The [IList](#) has a fixed size.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
}
```

```
public void Clear()
{
    _count = 0;
}

public bool Contains(object value)
{
    bool inList = false;
    for (int i = 0; i < Count; i++)
    {
        if (_contents[i] == value)
        {
            inList = true;
            break;
        }
    }
    return inList;
}

public int IndexOf(object value)
{
    int itemIndex = -1;
    for (int i = 0; i < Count; i++)
    {
        if (_contents[i] == value)
        {
            itemIndex = i;
            break;
        }
    }
    return itemIndex;
}

public void Insert(int index, object value)
{
    if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
    {
        _count++;

        for (int i = Count - 1; i > index; i--)
        {
            _contents[i] = _contents[i - 1];
        }
        _contents[index] = value;
    }
}

public bool IsFixedSize
{
    get
    {
        return true;
    }
}

public bool IsReadOnly
{
    get
    {
        return false;
    }
}

public void Remove(object value)
{
    RemoveAt(IndexOf(value));
}
```

```

        RemoveAt(index);
    }

    public void RemoveAt(int index)
    {
        if ((index >= 0) && (index < Count))
        {
            for (int i = index; i < Count - 1; i++)
            {
                _contents[i] = _contents[i + 1];
            }
            _count--;
        }
    }

    public object this[int index]
    {
        get
        {
            return _contents[index];
        }
        set
        {
            _contents[index] = value;
        }
    }

    // ICollection Members

    public void CopyTo(Array array, int index)
    {
        int j = index;
        for (int i = 0; i < Count; i++)
        {
            array.SetValue(_contents[i], j);
            j++;
        }
    }

    public int Count
    {
        get
        {
            return _count;
        }
    }

    public bool IsSynchronized
    {
        get
        {
            return false;
        }
    }

    // Return the current instance since the underlying store is not
    // publicly available.
    public object SyncRoot
    {
        get
        {
            return this;
        }
    }

    // IEnumerable Members

```

```
public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",
_contents.Length, _count);
    Console.Write("List contents:");
    for (int i = 0; i < Count; i++)
    {
        Console.Write(" {0}", _contents[i]);
    }
    Console.WriteLine();
}
```

# IList.Clear

## In this Article

Removes all items from the [IList](#).

```
public void Clear ();  
abstract member Clear : unit -> unit
```

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [IList](#) is read-only.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
  
    public void Clear()  
    {  
        _count = 0;  
    }  
  
    public bool Contains(object value)  
    {  
        bool inList = false;  
        for (int i = 0; i < Count; i++)  
        {  
            if (_contents[i] == value)  
            {  
                inList = true;  
                break;  
            }  
        }  
        return inList;  
    }  
}
```

```
}

public int IndexOf(object value)
{
    int itemIndex = -1;
    for (int i = 0; i < Count; i++)
    {
        if (_contents[i] == value)
        {
            itemIndex = i;
            break;
        }
    }
    return itemIndex;
}

public void Insert(int index, object value)
{
    if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
    {
        _count++;

        for (int i = Count - 1; i > index; i--)
        {
            _contents[i] = _contents[i - 1];
        }
        _contents[index] = value;
    }
}

public bool IsFixedSize
{
    get
    {
        return true;
    }
}

public bool IsReadOnly
{
    get
    {
        return false;
    }
}

public void Remove(object value)
{
    RemoveAt(IndexOf(value));
}

public void RemoveAt(int index)
{
    if ((index >= 0) && (index < Count))
    {
        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
        _count--;
    }
}

public object this[int index]
{
```

```

        get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get
    {
        return this;
    }
}

// IEnumerable Members

public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",
_contents.Length, _count);
    Console.Write("List contents:");
    for (int i = 0; i < Count; i++)
    {
        Console.Write(" {0}", _contents[i]);
    }
}

```

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

## Remarks

Implementations of this method can vary in how they handle the [ICollection.Count](#) and the capacity of a collection. Typically, the count is set to zero, and references to other objects from elements of the collection are also released. The capacity can be set to zero or a default value, or it can remain unchanged.

# IList.Contains

## In this Article

Determines whether the [IList](#) contains a specific value.

```
public bool Contains (object value);  
abstract member Contains : obj -> bool
```

### Parameters

value Object Object

The object to locate in the [IList](#).

### Returns

[Boolean](#) Boolean

`true` if the [Object](#) is found in the [IList](#); otherwise, `false`.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
  
    public void Clear()  
    {  
        _count = 0;  
    }  
  
    public bool Contains(object value)  
    {  
        bool inList = false;  
        for (int i = 0; i < Count; i++)  
        {  
            if (contents[i] == value)
```

```

        {
            inList = true;
            break;
        }
    }
    return inList;
}

public int IndexOf(object value)
{
    int itemIndex = -1;
    for (int i = 0; i < Count; i++)
    {
        if (_contents[i] == value)
        {
            itemIndex = i;
            break;
        }
    }
    return itemIndex;
}

public void Insert(int index, object value)
{
    if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
    {
        _count++;

        for (int i = Count - 1; i > index; i--)
        {
            _contents[i] = _contents[i - 1];
        }
        _contents[index] = value;
    }
}

public bool IsFixedSize
{
    get
    {
        return true;
    }
}

public bool IsReadOnly
{
    get
    {
        return false;
    }
}

public void Remove(object value)
{
    RemoveAt(IndexOf(value));
}

public void RemoveAt(int index)
{
    if ((index >= 0) && (index < Count))
    {
        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
    }
}

```

```

        _count--;
    }

}

public object this[int index]
{
    get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get
    {
        return this;
    }
}

// IEnumerable Members

public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of 50, and currently has {0} elements.", 

```

```
Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",  
    _contents.Length, _count);  
    Console.Write("List contents:");  
    for (int i = 0; i < Count; i++)  
    {  
        Console.Write(" {0}", _contents[i]);  
    }  
    Console.WriteLine();  
}
```

## Remarks

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

# IList.IndexOf

## In this Article

Determines the index of a specific item in the [IList](#).

```
public int IndexOf (object value);  
abstract member IndexOf : obj -> int
```

### Parameters

value Object Object

The object to locate in the [IList](#).

### Returns

[Int32](#) [Int32](#)

The index of `value` if found in the list; otherwise, -1.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
  
    public void Clear()  
    {  
        _count = 0;  
    }  
  
    public bool Contains(object value)  
    {  
        bool inList = false;  
        for (int i = 0; i < Count; i++)  
        {  
            if (_contents[i] == value)
```

```
        {
            inList = true;
            break;
        }
    }
    return inList;
}

public int IndexOf(object value)
{
    int itemIndex = -1;
    for (int i = 0; i < Count; i++)
    {
        if (_contents[i] == value)
        {
            itemIndex = i;
            break;
        }
    }
    return itemIndex;
}

public void Insert(int index, object value)
{
    if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
    {
        _count++;

        for (int i = Count - 1; i > index; i--)
        {
            _contents[i] = _contents[i - 1];
        }
        _contents[index] = value;
    }
}

public bool IsFixedSize
{
    get
    {
        return true;
    }
}

public bool IsReadOnly
{
    get
    {
        return false;
    }
}

public void Remove(object value)
{
    RemoveAt(IndexOf(value));
}

public void RemoveAt(int index)
{
    if ((index >= 0) && (index < Count))
    {
        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
    }
}
```

```

        _count--;
    }

}

public object this[int index]
{
    get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get
    {
        return this;
    }
}

// IEnumerable Members

public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of 50, and currently has {0} elements.", _contents.Length);
}

```

```
Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",  
    _contents.Length, _count);  
    Console.Write("List contents:");  
    for (int i = 0; i < Count; i++)  
    {  
        Console.Write(" {0}", _contents[i]);  
    }  
    Console.WriteLine();  
}
```

## Remarks

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

# IList.Insert

## In this Article

Inserts an item to the [IList](#) at the specified index.

```
public void Insert (int index, object value);  
abstract member Insert : int * obj -> unit
```

### Parameters

index	<a href="#">Int32</a>
-------	-----------------------

The zero-based index at which `value` should be inserted.

value	<a href="#">Object</a>
-------	------------------------

The object to insert into the [IList](#).

### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is not a valid index in the [IList](#).

[NotSupportedException](#) [NotSupportedException](#)

The [IList](#) is read-only.

-or-

The [IList](#) has a fixed size.

[NullReferenceException](#) [NullReferenceException](#)

`value` is null reference in the [IList](#).

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
    }  
}
```

```
        else
        {
            return -1;
        }
    }

    public void Clear()
    {
        _count = 0;
    }

    public bool Contains(object value)
    {
        bool inList = false;
        for (int i = 0; i < Count; i++)
        {
            if (_contents[i] == value)
            {
                inList = true;
                break;
            }
        }
        return inList;
    }

    public int IndexOf(object value)
    {
        int itemIndex = -1;
        for (int i = 0; i < Count; i++)
        {
            if (_contents[i] == value)
            {
                itemIndex = i;
                break;
            }
        }
        return itemIndex;
    }

    public void Insert(int index, object value)
    {
        if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
        {
            _count++;

            for (int i = Count - 1; i > index; i--)
            {
                _contents[i] = _contents[i - 1];
            }
            _contents[index] = value;
        }
    }

    public bool IsFixedSize
    {
        get
        {
            return true;
        }
    }

    public bool IsReadOnly
    {
        get
        {
            return false;
        }
    }
}
```

```

        return false;
    }

}

public void Remove(object value)
{
    RemoveAt(IndexOf(value));
}

public void RemoveAt(int index)
{
    if ((index >= 0) && (index < Count))
    {
        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
        _count--;
    }
}

public object this[int index]
{
    get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get

```

```
        {
            return this;
        }
    }

// IEnumerable Members

public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",
    _contents.Length, _count);
    Console.Write("List contents:");
    for (int i = 0; i < Count; i++)
    {
        Console.Write(" {0}", _contents[i]);
    }
    Console.WriteLine();
}
}
```

## Remarks

If `index` equals the number of items in the `IList`, then `value` is appended to the end.

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.

# IList.IsFixedSize

## In this Article

Gets a value indicating whether the [IList](#) has a fixed size.

```
public bool IsFixedSize { get; }  
member this.IsFixedSize : bool
```

Returns

[Boolean](#)

`true` if the [IList](#) has a fixed size; otherwise, `false`.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
  
    public void Clear()  
    {  
        _count = 0;  
    }  
  
    public bool Contains(object value)  
    {  
        bool inList = false;  
        for (int i = 0; i < Count; i++)  
        {  
            if (_contents[i] == value)  
            {  
                inList = true;  
                break;  
            }  
        }  
        return inList;  
    }  
}
```

```

}

public int IndexOf(object value)
{
    int itemIndex = -1;
    for (int i = 0; i < Count; i++)
    {
        if (_contents[i] == value)
        {
            itemIndex = i;
            break;
        }
    }
    return itemIndex;
}

public void Insert(int index, object value)
{
    if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
    {
        _count++;

        for (int i = Count - 1; i > index; i--)
        {
            _contents[i] = _contents[i - 1];
        }
        _contents[index] = value;
    }
}

public bool IsFixedSize
{
    get
    {
        return true;
    }
}

public bool IsReadOnly
{
    get
    {
        return false;
    }
}

public void Remove(object value)
{
    RemoveAt(IndexOf(value));
}

public void RemoveAt(int index)
{
    if ((index >= 0) && (index < Count))
    {
        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
        _count--;
    }
}

public object this[int index]
{

```

```

        get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get
    {
        return this;
    }
}

// IEnumerable Members

public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",
_contents.Length, _count);
    Console.Write("List contents:");
    for (int i = 0; i < Count; i++)
    {
        Console.Write(" {0}", _contents[i]);
    }
}

```

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

## 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.

# IList.IsReadOnly IList.IsReadOnly

## In this Article

Gets a value indicating whether the [IList](#) is read-only.

```
public bool IsReadOnly { get; }  
member this.IsReadOnly : bool
```

Returns

[Boolean](#) Boolean

[true](#) if the [IList](#) is read-only; otherwise, [false](#).

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
  
    public void Clear()  
    {  
        _count = 0;  
    }  
  
    public bool Contains(object value)  
    {  
        bool inList = false;  
        for (int i = 0; i < Count; i++)  
        {  
            if (_contents[i] == value)  
            {  
                inList = true;  
                break;  
            }  
        }  
        return inList;  
    }  
}
```

```
}

public int IndexOf(object value)
{
    int itemIndex = -1;
    for (int i = 0; i < Count; i++)
    {
        if (_contents[i] == value)
        {
            itemIndex = i;
            break;
        }
    }
    return itemIndex;
}

public void Insert(int index, object value)
{
    if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
    {
        _count++;

        for (int i = Count - 1; i > index; i--)
        {
            _contents[i] = _contents[i - 1];
        }
        _contents[index] = value;
    }
}

public bool IsFixedSize
{
    get
    {
        return true;
    }
}

public bool IsReadOnly
{
    get
    {
        return false;
    }
}

public void Remove(object value)
{
    RemoveAt(IndexOf(value));
}

public void RemoveAt(int index)
{
    if ((index >= 0) && (index < Count))
    {
        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
        _count--;
    }
}

public object this[int index]
{
```

```

        get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get
    {
        return this;
    }
}

// IEnumerable Members

public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",
_contents.Length, _count);
    Console.Write("List contents:");
    for (int i = 0; i < Count; i++)
    {
        Console.Write(" {0}", _contents[i]);
    }
}

```

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

## Remarks

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

# IList.Item[Int32] IList.Item[Int32]

## In this Article

Gets or sets the element at the specified index.

```
public object this[int index] { get; set; }

member this.Item(int) : obj with get, set
```

## Parameters

index	Int32 Int32
-------	-------------

The zero-based index of the element to get or set.

## Returns

[Object Object](#)

The element at the specified index.

## Exceptions

[ArgumentOutOfRangeException ArgumentOutOfRangeException](#)

`index` is not a valid index in the [IList](#).

[NotSupportedException NotSupportedException](#)

The property is set and the [IList](#) is read-only.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList
{
    private object[] _contents = new object[8];
    private int _count;

    public SimpleList()
    {
        _count = 0;
    }

    // IList Members
    public int Add(object value)
    {
        if (_count < _contents.Length)
        {
            _contents[_count] = value;
            _count++;

            return (_count - 1);
        }
        else
        {
            return -1;
        }
    }

    public void Clear()
    {
        _contents = new object[8];
        _count = 0;
    }

    public object this[int index]
    {
        get
        {
            if (index < 0 || index >= _count)
                throw new ArgumentOutOfRangeException("index");

            return _contents[index];
        }
        set
        {
            if (index < 0 || index >= _count)
                throw new ArgumentOutOfRangeException("index");

            _contents[index] = value;
        }
    }
}
```

```
{  
    _count = 0;  
}  
  
public bool Contains(object value)  
{  
    bool inList = false;  
    for (int i = 0; i < Count; i++)  
    {  
        if (_contents[i] == value)  
        {  
            inList = true;  
            break;  
        }  
    }  
    return inList;  
}  
  
public int IndexOf(object value)  
{  
    int itemIndex = -1;  
    for (int i = 0; i < Count; i++)  
    {  
        if (_contents[i] == value)  
        {  
            itemIndex = i;  
            break;  
        }  
    }  
    return itemIndex;  
}  
  
public void Insert(int index, object value)  
{  
    if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))  
    {  
        _count++;  
  
        for (int i = Count - 1; i > index; i--)  
        {  
            _contents[i] = _contents[i - 1];  
        }  
        _contents[index] = value;  
    }  
}  
  
public bool IsFixedSize  
{  
    get  
    {  
        return true;  
    }  
}  
  
public bool IsReadOnly  
{  
    get  
    {  
        return false;  
    }  
}  
  
public void Remove(object value)  
{  
    RemoveAt(IndexOf(value));  
}
```

```
        }

    public void RemoveAt(int index)
    {
        if ((index >= 0) && (index < Count))
        {
            for (int i = index; i < Count - 1; i++)
            {
                _contents[i] = _contents[i + 1];
            }
            _count--;
        }
    }

    public object this[int index]
    {
        get
        {
            return _contents[index];
        }
        set
        {
            _contents[index] = value;
        }
    }

// ICollection Members

    public void CopyTo(Array array, int index)
    {
        int j = index;
        for (int i = 0; i < Count; i++)
        {
            array.SetValue(_contents[i], j);
            j++;
        }
    }

    public int Count
    {
        get
        {
            return _count;
        }
    }

    public bool IsSynchronized
    {
        get
        {
            return false;
        }
    }

// Return the current instance since the underlying store is not
// publicly available.
    public object SyncRoot
    {
        get
        {
            return this;
        }
    }

// IEnumerable Members
```

```
public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",
    _contents.Length, _count);
    Console.Write("List contents:");
    for (int i = 0; i < Count; i++)
    {
        Console.Write(" {0}", _contents[i]);
    }
    Console.WriteLine();
}
```

## 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 keyword to define the indexers instead of implementing the [Item\[Int32\]](#) property. Visual Basic implements [Item\[Int32\]](#) as a default property, which provides the same indexing functionality.

# IList.Remove

## In this Article

Removes the first occurrence of a specific object from the [IList](#).

```
public void Remove (object value);  
abstract member Remove : obj -> unit
```

### Parameters

value Object Object

The object to remove from the [IList](#).

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [IList](#) is read-only.

-or-

The [IList](#) has a fixed size.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
  
    public void Clear()  
    {  
        _count = 0;  
    }  
  
    public bool Contains(object value)  
    {
```

```

        bool inList = false;
        for (int i = 0; i < Count; i++)
        {
            if (_contents[i] == value)
            {
                inList = true;
                break;
            }
        }
        return inList;
    }

    public int IndexOf(object value)
    {
        int itemIndex = -1;
        for (int i = 0; i < Count; i++)
        {
            if (_contents[i] == value)
            {
                itemIndex = i;
                break;
            }
        }
        return itemIndex;
    }

    public void Insert(int index, object value)
    {
        if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
        {
            _count++;

            for (int i = Count - 1; i > index; i--)
            {
                _contents[i] = _contents[i - 1];
            }
            _contents[index] = value;
        }
    }

    public bool IsFixedSize
    {
        get
        {
            return true;
        }
    }

    public bool IsReadOnly
    {
        get
        {
            return false;
        }
    }

    public void Remove(object value)
    {
        RemoveAt(IndexOf(value));
    }

    public void RemoveAt(int index)
    {
        if ((index >= 0) && (index < Count))
        {

```

```

        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
        _count--;
    }
}

public object this[int index]
{
    get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get
    {
        return this;
    }
}

// IEnumerable Members

public IEnumerator GetEnumerator()
{
    // Refer to the IEnumerator documentation for an example of
    // implementing an enumerator.
    throw new Exception("The method or operation is not implemented.");
}

```

```
}

public void PrintContents()
{
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",
    _contents.Length, _count);
    Console.Write("List contents:");
    for (int i = 0; i < Count; i++)
    {
        Console.Write(" {0}", _contents[i]);
    }
    Console.WriteLine();
}
```

## 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. If `value` is not found in the `IList`, the `IList` remains unchanged and no exception is thrown.

# IList.RemoveAt

## In this Article

Removes the [IList](#) item at the specified index.

```
public void RemoveAt (int index);  
abstract member RemoveAt : int -> unit
```

### Parameters

index	<a href="#">Int32</a>
-------	-----------------------

The zero-based index of the item to remove.

### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is not a valid index in the [IList](#).

[NotSupportedException](#) [NotSupportedException](#)

The [IList](#) is read-only.

-or-

The [IList](#) has a fixed size.

## Examples

The following example demonstrates the implementation of the [IList](#) interface to create a simple list, fixed-size list. This code is part of a larger example for the [IList](#) interface.

```
class SimpleList : IList  
{  
    private object[] _contents = new object[8];  
    private int _count;  
  
    public SimpleList()  
    {  
        _count = 0;  
    }  
  
    // IList Members  
    public int Add(object value)  
    {  
        if (_count < _contents.Length)  
        {  
            _contents[_count] = value;  
            _count++;  
  
            return (_count - 1);  
        }  
        else  
        {  
            return -1;  
        }  
    }  
  
    public void Clear()  
    {  
        _count = 0;  
    }  
}
```

```

        _count = v,
    }

    public bool Contains(object value)
    {
        bool inList = false;
        for (int i = 0; i < Count; i++)
        {
            if (_contents[i] == value)
            {
                inList = true;
                break;
            }
        }
        return inList;
    }

    public int IndexOf(object value)
    {
        int itemIndex = -1;
        for (int i = 0; i < Count; i++)
        {
            if (_contents[i] == value)
            {
                itemIndex = i;
                break;
            }
        }
        return itemIndex;
    }

    public void Insert(int index, object value)
    {
        if ((_count + 1 <= _contents.Length) && (index < Count) && (index >= 0))
        {
            _count++;

            for (int i = Count - 1; i > index; i--)
            {
                _contents[i] = _contents[i - 1];
            }
            _contents[index] = value;
        }
    }

    public bool IsFixedSize
    {
        get
        {
            return true;
        }
    }

    public bool IsReadOnly
    {
        get
        {
            return false;
        }
    }

    public void Remove(object value)
    {
        RemoveAt(IndexOf(value));
    }
}

```

```
public void RemoveAt(int index)
{
    if ((index >= 0) && (index < Count))
    {
        for (int i = index; i < Count - 1; i++)
        {
            _contents[i] = _contents[i + 1];
        }
        _count--;
    }
}

public object this[int index]
{
    get
    {
        return _contents[index];
    }
    set
    {
        _contents[index] = value;
    }
}

// ICollection Members

public void CopyTo(Array array, int index)
{
    int j = index;
    for (int i = 0; i < Count; i++)
    {
        array.SetValue(_contents[i], j);
        j++;
    }
}

public int Count
{
    get
    {
        return _count;
    }
}

public bool IsSynchronized
{
    get
    {
        return false;
    }
}

// Return the current instance since the underlying store is not
// publicly available.
public object SyncRoot
{
    get
    {
        return this;
    }
}

// IEnumerable Members

public IEnumerator GetEnumerator()
```

```
{  
    // Refer to the IEnumator documentation for an example of  
    // implementing an enumerator.  
    throw new Exception("The method or operation is not implemented.");  
}  
  
public void PrintContents()  
{  
    Console.WriteLine("List has a capacity of {0} and currently has {1} elements.",  
    _contents.Length, _count);  
    Console.Write("List contents:");  
    for (int i = 0; i < Count; i++)  
    {  
        Console.Write(" {0}", _contents[i]);  
    }  
    Console.WriteLine();  
}
```

## 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.

# IStructuralComparable IStructuralComparable Interface

Supports the structural comparison of collection objects.

## Declaration

```
public interface IStructuralComparable  
type IStructuralComparable = interface
```

## Inheritance Hierarchy

None

## Remarks

The [IStructuralComparable](#) interface enables you to implement customized comparisons for collection members. That is, you can define precisely what it means for one collection object to precede, follow, or occur in the same position in the sort order as a second collection object. You can then specify that this definition be used with a collection type that accepts the [IStructuralComparable](#) interface.

The interface has a single member, [CompareTo](#), which determines whether the current collection object is less than, equal to, or greater than a second object in the sort order. The actual comparison of the members or elements in the current instance with those in a second object is performed by an [IComparer](#) interface implementation, which contains the definition of your custom comparison.

### Note

The [IStructuralComparable](#) interface supports only structural comparisons for sorting or ordering. The [IStructuralEquatable](#) interface supports custom comparisons for structural equality.

The .NET Framework provides two default comparers. One is returned by the [StructuralComparisons.StructuralComparer](#) property; the other is returned by the [Comparer<T>.Default](#) property.

The generic tuple classes ([Tuple<T1>](#), [Tuple<T1,T2>](#), [Tuple<T1,T2,T3>](#), and so on) and the [Array](#) class provide explicit implementations of the [IStructuralComparable](#) interface. By casting (in C#) or converting (in Visual Basic) the current instance of an array or tuple to an [IStructuralComparable](#) interface value and providing your [IComparer](#) implementation as an argument to the [CompareTo](#) method, you can define a custom sort order for the array or collection. However, you do not call the [CompareTo](#) method directly in most cases. Instead, the [CompareTo](#) method is called by sorting methods such as [Sort\(Array, IComparer\)](#). In this case, you define your [IComparer](#) implementation and pass it as an argument to a sorting method or collection object's class constructor. The [CompareTo](#) method with your custom comparer is then called automatically whenever the collection is sorted.

## Methods

[CompareTo\(Object, IComparer\)](#)

[CompareTo\(Object, IComparer\)](#)

Determines whether the current collection object precedes, occurs in the same position as, or follows another object in the sort order.

# IComparable.CompareTo IComparable.CompareTo

## In this Article

Determines whether the current collection object precedes, occurs in the same position as, or follows another object in the sort order.

```
public int CompareTo (object other, System.Collections.IComparer comparer);  
abstract member CompareTo : obj * System.Collections.IComparer -> int
```

## Parameters

other Object Object

The object to compare with the current instance.

comparer IComparer IComparer

An object that compares members of the current collection object with the corresponding members of `other`.

## Returns

`Int32 Int32`

A signed integer that indicates the relationship of the current collection object to `other` in the sort order:

- If less than 0, the current instance precedes `other`.
- If 0, the current instance and `other` are equal.
- If greater than 0, the current instance follows `other`.

RETURN VALUE	DESCRIPTION
-1	The current instance precedes <code>other</code> .
0	The current instance and <code>other</code> are equal.
1	The current instance follows <code>other</code> .

## Exceptions

[ArgumentException](#) [ArgumentException](#)

This instance and `other` are not the same type.

## Examples

The following example creates an array of `Tuple<T1,T2,T3,T4,T5,T6>` objects that contains population data for three U.S. cities from 1960 to 2000. The sextuple's first component is the city name. The remaining five components represent the population at ten-year intervals from 1960 to 2000.

The `PopulationComparer` class provides an `IComparer` implementation that allows the array of sextuples to be sorted by any one of its components. Two values are provided to the `PopulationComparer` class in its constructor: the position of the component that defines the sort order, and a `Boolean` value that indicates whether the tuple objects should be sorted in ascending or descending order.

The example then displays the elements in the array in unsorted order, sorts them by the third component (the population in 1970) and displays them, and then sorts them by the sixth component (the population in 2000) and

displays them. Note that the example does not directly call the [IStructuralComparable.CompareTo](#) implementation. The method is called implicitly by the [Sort\(Array, IComparer\)](#) method for each tuple object in the array.

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

public class PopulationComparer<T1, T2, T3, T4, T5, T6> : IComparer
{
    private int itemPosition;
    private int multiplier = -1;

    public PopulationComparer(int component) : this(component, true)
    { }

    public PopulationComparer(int component, bool descending)
    {
        if (!descending) multiplier = 1;

        if (component <= 0 || component > 6)
            throw new ArgumentException("The component argument is out of range.");

        itemPosition = component;
    }

    public int Compare(object x, object y)
    {
        var tX = x as Tuple<T1, T2, T3, T4, T5, T6>;
        if (tX == null)
        {
            return 0;
        }
        else
        {
            var tY = y as Tuple<T1, T2, T3, T4, T5, T6>;
            switch (itemPosition)
            {
                case 1:
                    return Comparer<T1>.Default.Compare(tX.Item1, tY.Item1) * multiplier;
                case 2:
                    return Comparer<T2>.Default.Compare(tX.Item2, tY.Item2) * multiplier;
                case 3:
                    return Comparer<T3>.Default.Compare(tX.Item3, tY.Item3) * multiplier;
                case 4:
                    return Comparer<T4>.Default.Compare(tX.Item4, tY.Item4) * multiplier;
                case 5:
                    return Comparer<T5>.Default.Compare(tX.Item5, tY.Item5) * multiplier;
                case 6:
                    return Comparer<T6>.Default.Compare(tX.Item6, tY.Item6) * multiplier;
                default:
                    return Comparer<T1>.Default.Compare(tX.Item1, tY.Item1) * multiplier;
            }
        }
    }
}

public class Example
{
    public static void Main()
    {
        // Create array of sextuple with population data for three U.S.
        // cities, 1960-2000.
        Tuple<string, int, int, int, int, int>[] cities =
            { Tuple.Create("Los Angeles", 2479015, 2816061, 2966850, 3485398, 3694820),
              Tuple.Create("New York", 7781984, 7894862, 7071639, 7322564, 8008278),
            };
    }
}
```

```

        Tuple.Create("Chicago", 3550904, 3366957, 3005072, 2783726, 2896016) };

    // Display array in unsorted order.
    Console.WriteLine("In unsorted order:");
    foreach (var city in cities)
        Console.WriteLine(city.ToString());
    Console.WriteLine();

    Array.Sort(cities, new PopulationComparer<string, int, int, int, int, int>(3));

    // Display array in sorted order.
    Console.WriteLine("Sorted by population in 1970:");
    foreach (var city in cities)
        Console.WriteLine(city.ToString());
    Console.WriteLine();

    Array.Sort(cities, new PopulationComparer<string, int, int, int, int, int>(6));

    // Display array in sorted order.
    Console.WriteLine("Sorted by population in 2000:");
    foreach (var city in cities)
        Console.WriteLine(city.ToString());
}

// The example displays the following output:
// In unsorted order:
// (Los Angeles, 2479015, 2816061, 2966850, 3485398, 3694820)
// (New York, 7781984, 7894862, 7071639, 7322564, 8008278)
// (Chicago, 3550904, 3366957, 3005072, 2783726, 2896016)
//
// Sorted by population in 1970:
// (New York, 7781984, 7894862, 7071639, 7322564, 8008278)
// (Chicago, 3550904, 3366957, 3005072, 2783726, 2896016)
// (Los Angeles, 2479015, 2816061, 2966850, 3485398, 3694820)
//
// Sorted by population in 2000:
// (New York, 7781984, 7894862, 7071639, 7322564, 8008278)
// (Los Angeles, 2479015, 2816061, 2966850, 3485398, 3694820)
// (Chicago, 3550904, 3366957, 3005072, 2783726, 2896016)

```

## Remarks

The [CompareTo](#) method supports custom structural comparison and sorting of array and tuple objects. The [CompareTo](#) method calls the `comparer` object's [IComparer.Compare](#) method to compare individual array elements or tuple components, starting with the first element or component. The individual calls to [IComparer.Compare](#) end and the [CompareTo](#) method returns a value when one of the following conditions becomes true:

- The [IComparer.Compare](#) method returns -1.
- The [IComparer.Compare](#) method returns 1.
- The [IComparer.Compare](#) method is called for the last element or component in the collection object.

See

[IComparer](#)[IComparer](#)

Also

# IStructuralEquatable IStructuralEquatable Interface

Defines methods to support the comparison of objects for structural equality.

## Declaration

```
public interface IStructuralEquatable  
type IStructuralEquatable = interface
```

## Inheritance Hierarchy

None

## Remarks

Structural equality means that two objects are equal because they have equal values. It differs from reference equality, which indicates that two object references are equal because they reference the same physical object. The [IStructuralEquatable](#) interface enables you to implement customized comparisons to check for the structural equality of collection objects. That is, you can create your own definition of structural equality and specify that this definition be used with a collection type that accepts the [IStructuralEquatable](#) interface. The interface has two members: [Equals](#), which tests for equality by using a specified [IEqualityComparer](#) implementation, and [GetHashCode](#), which returns identical hash codes for objects that are equal.

**Note**

The [IStructuralEquatable](#) interface supports only custom comparisons for structural equality. The [IComparable](#) interface supports custom structural comparisons for sorting and ordering.

The .NET Framework also provides default equality comparers, which are returned by the [EqualityComparer<T>.Default](#) and [StructuralComparisons.StructuralEqualityComparer](#) properties. For more information, see the example.

The generic tuple classes ([Tuple<T1>](#), [Tuple<T1,T2>](#), [Tuple<T1,T2,T3>](#), and so on) and the [Array](#) class provide explicit implementations of the [IStructuralEquatable](#) interface. By casting (in C#) or converting (in Visual Basic) the current instance of an array or tuple to an [IStructuralEquatable](#) interface value and providing your [IEqualityComparer](#) implementation as an argument to the [Equals](#) method, you can define a custom equality comparison for the array or collection.

## Methods

[Equals\(Object, IEqualityComparer\)](#)

[Equals\(Object, IEqualityComparer\)](#)

Determines whether an object is structurally equal to the current instance.

[GetHashCode\(IEqualityComparer\)](#)

[GetHashCode\(IEqualityComparer\)](#)

Returns a hash code for the current instance.

## See Also

[IEqualityComparer](#) [IEqualityComparer](#)



# IStructuralEquatable.Equals IStructuralEquatable.Equals

## In this Article

Determines whether an object is structurally equal to the current instance.

```
public bool Equals (object other, System.Collections.IEqualityComparer comparer);  
abstract member Equals : obj * System.Collections.IEqualityComparer -> bool
```

### Parameters

other Object Object

The object to compare with the current instance.

comparer IEqualityComparer IEqualityComparer

An object that determines whether the current instance and `other` are equal.

### Returns

Boolean Boolean

`true` if the two objects are equal; otherwise, `false`.

## Examples

The default equality comparer, `EqualityComparer<Object>.Default.Equals`, considers two `NaN` values to be equal. However, in some cases, you may want the comparison of `NaN` values for equality to return `false`, which indicates that the values cannot be compared. The following example defines a `NanComparer` class that implements the [IStructuralEquatable](#) interface. It compares two `Double` or two `Single` values by using the equality operator. It passes values of any other type to the default equality comparer.

```
using System;  
using System.Collections;  
using System.Collections.Generic;  
  
public class NanComparer : IEqualityComparer  
{  
    public new bool Equals(object x, object y)  
    {  
        if (x is float)  
            return (float) x == (float) y;  
        else if (x is double)  
            return (double) x == (double) y;  
        else  
            return EqualityComparer<object>.Default.Equals(x, y);  
    }  
  
    public int GetHashCode(object obj)  
    {  
        return EqualityComparer<object>.Default.GetHashCode(obj);  
    }  
}
```

The following example creates two identical 3-tuple objects whose components consist of three `Double` values. The value of the second component is `Double.NaN`. The example then calls the `Tuple<T1,T2,T3>.Equals` method, and it calls the [IStructuralEquatable.Equals](#) method three times. The first time, it passes the default equality comparer that is returned by the `EqualityComparer<T>.Default` property. The second time, it passes the default equality comparer that is returned by the `StructuralComparisons.StructuralEqualityComparer` property. The third time, it passes the custom

`NanComparer` object. As the output from the example shows, the first three method calls return `true`, whereas the fourth call returns `false`.

```
public class Example
{
    public static void Main()
    {
        var t1 = Tuple.Create(12.3, Double.NaN, 16.4);
        var t2 = Tuple.Create(12.3, Double.NaN, 16.4);

        // Call default Equals method.
        Console.WriteLine(t1.Equals(t2));

        IStructuralEquatable equ = t1;
        // Call IStructuralEquatable.Equals using default comparer.
        Console.WriteLine(equ.Equals(t2, EqualityComparer<object>.Default));

        // Call IStructuralEquatable.Equals using
        // StructuralComparisons.StructuralEqualityComparer.
        Console.WriteLine(equ.Equals(t2,
            StructuralComparisons.StructuralEqualityComparer));

        // Call IStructuralEquatable.Equals using custom comparer.
        Console.WriteLine(equ.Equals(t2, new NanComparer()));
    }
}

// The example displays the following output:
//      True
//      True
//      True
//      False
```

## Remarks

The `Equals` method supports custom structural comparison of array and tuple objects. This method in turn calls the `comparer` object's `IEqualityComparer.Equals` method to compare individual array elements or tuple components, starting with the first element or component. The individual calls to `IEqualityComparer.Equals` end and the `IStructuralEquatable.Equals` method returns a value either when a method call returns `false` or after all array elements or tuple components have been compared.

# IStructuralEquatable.GetHashCode IStructuralEquatable.GetHashCode

## In this Article

Returns a hash code for the current instance.

```
public int GetHashCode (System.Collections.IEqualityComparer comparer);  
abstract member GetHashCode : System.Collections.IEqualityComparer -> int
```

### Parameters

comparer IEqualityComparer | EqualityComparer

An object that computes the hash code of the current object.

### Returns

[Int32](#) [Int32](#)

The hash code for the current instance.

## Remarks

Implement this method to return customized hash codes for collection objects that correspond to the customized comparison for structural equality provided by the [Equals](#) method.

# Queue Queue Class

Represents a first-in, first-out collection of objects.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public class Queue : ICloneable, System.Collections.ICollection

type Queue = class
    interface ICollection
    interface ICloneable
    interface IEnumerable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

This class implements a queue as a circular array. Objects stored in a [Queue](#) are inserted at one end and removed from the other.

**Important**

We don't recommend that you use the [Queue](#) class for new development. Instead, we recommend that you use the generic [Queue<T>](#) class. For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

Queues and stacks are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use [Queue](#) if you need to access the information in the same order that it is stored in the collection. Use [Stack](#) if you need to access the information in reverse order. Use [ConcurrentQueue<T>](#) or [ConcurrentStack<T>](#) if you need to access the collection from multiple threads concurrently.

Three main operations can be performed on a [Queue](#) and its elements:

- [Enqueue](#) adds an element to the end of the [Queue](#).
- [Dequeue](#) removes the oldest element from the start of the [Queue](#).
- [Peek](#) returns the oldest element that is at the start of the [Queue](#) but does not remove it from the [Queue](#).

The capacity of a [Queue](#) is the number of elements the [Queue](#) can hold. As elements are added to a [Queue](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#).

The growth factor is the number by which the current capacity is multiplied when a greater capacity is required. The growth factor is determined when the [Queue](#) is constructed. The default growth factor is 2.0. The capacity of the [Queue](#) will always increase by at least a minimum of four, regardless of the growth factor. For example, a [Queue](#) with a growth factor of 1.0 will always increase in capacity by four when a greater capacity is required.

[Queue](#) accepts [null](#) as a valid value and allows duplicate elements.

For the generic version of this collection, see [System.Collections.Generic.Queue<T>](#)

## Constructors

[Queue\(\)](#)

`Queue()`

Initializes a new instance of the [Queue](#) class that is empty, has the default initial capacity, and uses the default growth factor.

`Queue(ICollection)`

`Queue(ICollection)`

Initializes a new instance of the [Queue](#) class that contains elements copied from the specified collection, has the same initial capacity as the number of elements copied, and uses the default growth factor.

`Queue(Int32)`

`Queue(Int32)`

Initializes a new instance of the [Queue](#) class that is empty, has the specified initial capacity, and uses the default growth factor.

`Queue(Int32, Single)`

`Queue(Int32, Single)`

Initializes a new instance of the [Queue](#) class that is empty, has the specified initial capacity, and uses the specified growth factor.

## Properties

`Count`

`Count`

Gets the number of elements contained in the [Queue](#).

`IsSynchronized`

`IsSynchronized`

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

`SyncRoot`

`SyncRoot`

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

## Methods

`Clear()`

`Clear()`

Removes all objects from the [Queue](#).

`Clone()`

`Clone()`

Creates a shallow copy of the [Queue](#).

`Contains(Object)`

`Contains(Object)`

Determines whether an element is in the [Queue](#).

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

Copies the [Queue](#) elements to an existing one-dimensional [Array](#), starting at the specified array index.

`Dequeue()`

`Dequeue()`

Removes and returns the object at the beginning of the [Queue](#).

`Enqueue(Object)`

`Enqueue(Object)`

Adds an object to the end of the [Queue](#).

`GetEnumerator()`

`GetEnumerator()`

Returns an enumerator that iterates through the [Queue](#).

`Peek()`

`Peek()`

Returns the object at the beginning of the [Queue](#) without removing it.

`Synchronized(Queue)`

`Synchronized(Queue)`

Returns a new [Queue](#) that wraps the original queue, and is thread safe.

`ToArray()`

`ToArray()`

Copies the [Queue](#) elements to a new array.

`TrimToSize()`

`TrimToSize()`

Sets the capacity to the actual number of elements in the [Queue](#).

## 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.

To guarantee the thread safety of the [Queue](#), all operations must be done through the wrapper returned by the [Synchronized\(Queue\)](#) method.

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 causes 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.

# Queue.Clear Queue.Clear

## In this Article

Removes all objects from the [Queue](#).

```
public virtual void Clear ();  
  
abstract member Clear : unit -> unit  
override this.Clear : unit -> unit
```

## Examples

The following example shows how to clear the values of the [Queue](#).

```

using System;
using System.Collections;
public class SamplesQueue {
    public static void Main() {
        // Creates and initializes a new Queue.
        Queue myQ = new Queue();
        myQ.Enqueue( "The" );
        myQ.Enqueue( "quick" );
        myQ.Enqueue( "brown" );
        myQ.Enqueue( "fox" );
        myQ.Enqueue( "jumps" );

        // Displays the count and values of the Queue.
        Console.WriteLine( "Initially," );
        Console.WriteLine( "    Count      : {0}", myQ.Count );
        Console.Write( "    Values:" );
        PrintValues( myQ );

        // Clears the Queue.
        myQ.Clear();

        // Displays the count and values of the Queue.
        Console.WriteLine( "After Clear," );
        Console.WriteLine( "    Count      : {0}", myQ.Count );
        Console.Write( "    Values:" );
        PrintValues( myQ );
    }

    public static void PrintValues( Queue myQ ) {
        foreach ( Object myObj in myQ ) {
            Console.Write( "    {0}", myObj );
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Initially,
Count      : 5
Values:    The    quick    brown    fox    jumps
After Clear,
Count      : 0
Values:
*/

```

## 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](#), call [TrimToSize](#). Trimming an empty [Queue](#) sets the capacity of the [Queue](#) to the default capacity.

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

See

[TrimToSize\(\)](#)[TrimToSize\(\)](#)

Also

# Queue.Clone Queue.Clone

## In this Article

Creates a shallow copy of the [Queue](#).

```
public virtual object Clone ();  
  
abstract member Clone : unit -> obj  
override this.Clone : unit -> obj
```

Returns

[Object Object](#)

A shallow copy of the [Queue](#).

## Remarks

A shallow copy of a collection copies only the elements of the collection, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new collection point to the same objects that the references in the original collection point to.

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](#).

# Queue.Contains Queue.Contains

## In this Article

Determines whether an element is in the [Queue](#).

```
public virtual bool Contains (object obj);  
  
abstract member Contains : obj -> bool  
override this.Contains : obj -> bool
```

## Parameters

obj [Object](#) [Object](#)

The [Object](#) to locate in the [Queue](#). The value can be `null`.

## Returns

[Boolean](#) [Boolean](#)

`true` if `obj` is found in the [Queue](#); otherwise, `false`.

## Remarks

This method determines equality by calling [Object.Equals](#).

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where `n` is [Count](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `obj` to determine whether `item` exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `obj` parameter on the objects in the collection.

See

[Performing Culture-Insensitive String Operations](#)

Also

# Queue.CopyTo Queue.CopyTo

## In this Article

Copies the [Queue](#) elements to an existing one-dimensional [Array](#), starting at the specified array index.

```
public virtual void CopyTo (Array array, int index);  
  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

## Parameters

array [Array](#) [Array](#)

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

index [Int32](#) [Int32](#)

The zero-based index in `array` at which copying begins.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`array` is `null`.

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is less than zero.

[ArgumentException](#) [ArgumentException](#)

`array` is multidimensional.

-or-

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

[ArrayTypeMismatchException](#) [ArrayTypeMismatchException](#)

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

## Examples

The following example shows how to copy a [Queue](#) into a one-dimensional array.

```

using System;
using System.Collections;
public class SamplesQueue {

    public static void Main() {

        // Creates and initializes the source Queue.
        Queue mySourceQ = new Queue();
        mySourceQ.Enqueue( "three" );
        mySourceQ.Enqueue( "napping" );
        mySourceQ.Enqueue( "cats" );
        mySourceQ.Enqueue( "in" );
        mySourceQ.Enqueue( "the" );
        mySourceQ.Enqueue( "barn" );

        // Creates and initializes the one-dimensional target Array.
        Array myTargetArray=Array.CreateInstance( typeof(String), 15 );
        myTargetArray.SetValue( "The", 0 );
        myTargetArray.SetValue( "quick", 1 );
        myTargetArray.SetValue( "brown", 2 );
        myTargetArray.SetValue( "fox", 3 );
        myTargetArray.SetValue( "jumps", 4 );
        myTargetArray.SetValue( "over", 5 );
        myTargetArray.SetValue( "the", 6 );
        myTargetArray.SetValue( "lazy", 7 );
        myTargetArray.SetValue( "dog", 8 );

        // Displays the values of the target Array.
        Console.WriteLine( "The target Array contains the following (before and after copying):" );
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source Queue to the target Array, starting at index 6.
        mySourceQ.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source Queue to a new standard array.
        Object[] myStandardArray = mySourceQ.ToArray();

        // Displays the values of the new standard array.
        Console.WriteLine( "The new standard array contains the following:" );
        PrintValues( myStandardArray, ' ' );
    }

    public static void PrintValues( Array myArr, char mySeparator ) {
        foreach ( Object myObj in myArr ) {
            Console.Write( "{0}{1}", mySeparator, myObj );
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The target Array contains the following (before and after copying):
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over three napping cats in the barn
The new standard array contains the following:
three napping cats in the barn
*/

```

## Remarks

The elements are copied to the [Array](#) in the same order in which the enumerator iterates through the [Queue](#).

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

# Queue.Count Queue.Count

## In this Article

Gets the number of elements contained in the [Queue](#).

```
public virtual int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [Queue](#).

## Remarks

The capacity of a [Queue](#) is the number of elements that the [Queue](#) can store. [Count](#) is the number of elements that are actually in the [Queue](#).

The capacity of a [Queue](#) is always greater than or equal to [Count](#). If [Count](#) exceeds the capacity while adding elements, the capacity is automatically increased by reallocating the internal array before copying the old elements and adding the new elements. The new capacity is determined by multiplying the current capacity by the growth factor, which is determined when the [Queue](#) is constructed. The capacity of the [Queue](#) will always increase by a minimum value, regardless of the growth factor; a growth factor of 1.0 will not prevent the [Queue](#) from increasing in size.

The capacity can be decreased by calling [TrimToSize](#).

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

# Queue.Dequeue Queue.Dequeue

## In this Article

Removes and returns the object at the beginning of the [Queue](#).

```
public virtual object Dequeue ();  
  
abstract member Dequeue : unit -> obj  
override this.Dequeue : unit -> obj
```

Returns

[Object Object](#)

The object that is removed from the beginning of the [Queue](#).

Exceptions

[InvalidOperationException InvalidOperationException](#)

The [Queue](#) is empty.

## Examples

The following example shows how to add elements to the [Queue](#), remove elements from the [Queue](#), or view the element at the beginning of the [Queue](#).

```

using System;
using System.Collections;
public class SamplesQueue {

    public static void Main() {

        // Creates and initializes a new Queue.
        Queue myQ = new Queue();
        myQ.Enqueue( "The" );
        myQ.Enqueue( "quick" );
        myQ.Enqueue( "brown" );
        myQ.Enqueue( "fox" );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Removes an element from the Queue.
        Console.WriteLine( "(Dequeue) {0}", myQ.Dequeue() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Removes another element from the Queue.
        Console.WriteLine( "(Dequeue) {0}", myQ.Dequeue() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Views the first element in the Queue but does not remove it.
        Console.WriteLine( "(Peek) {0}", myQ.Peek() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );
    }

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

/*
This code produces the following output.

Queue values: The quick brown fox
(Dequeue) The
Queue values: quick brown fox
(Dequeue) quick
Queue values: brown fox
(Peek) brown
Queue values: brown fox

*/

```

## Remarks

This method is similar to the [Peek](#) method, but [Peek](#) does not modify the [Queue](#).

[null](#) can be added to the [Queue](#) as a value. To distinguish between a null value and the end of the [Queue](#), check the

[Count](#) property or catch the [InvalidOperationException](#), which is thrown when the [Queue](#) is empty.

This method is an O(1) operation.

See

[Enqueue\(Object\)](#)

Also

[Peek\(\)](#)

[Peek\(\)](#)

# Queue.Enqueue Queue.Enqueue

## In this Article

Adds an object to the end of the [Queue](#).

```
public virtual void Enqueue (object obj);  
  
abstract member Enqueue : obj -> unit  
override this.Enqueue : obj -> unit
```

## Parameters

obj	<a href="#">Object Object</a>
-----	-------------------------------

The object to add to the [Queue](#). The value can be `null`.

## Examples

The following example shows how to add elements to the [Queue](#), remove elements from the [Queue](#), or view the element at the beginning of the [Queue](#).

```

using System;
using System.Collections;
public class SamplesQueue {

    public static void Main() {

        // Creates and initializes a new Queue.
        Queue myQ = new Queue();
        myQ.Enqueue( "The" );
        myQ.Enqueue( "quick" );
        myQ.Enqueue( "brown" );
        myQ.Enqueue( "fox" );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Removes an element from the Queue.
        Console.WriteLine( "(Dequeue) {0}", myQ.Dequeue() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Removes another element from the Queue.
        Console.WriteLine( "(Dequeue) {0}", myQ.Dequeue() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Views the first element in the Queue but does not remove it.
        Console.WriteLine( "(Peek) {0}", myQ.Peek() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );
    }

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

/*
This code produces the following output.

Queue values: The quick brown fox
(Dequeue) The
Queue values: quick brown fox
(Dequeue) quick
Queue values: brown fox
(Peek) brown
Queue values: brown fox

*/

```

## Remarks

The capacity of a [Queue](#) is the number of elements the [Queue](#) can hold. As elements are added to a [Queue](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#).

The growth factor is the number by which the current capacity is multiplied when a greater capacity is required. The growth factor is determined when the [Queue](#) is constructed. The capacity of the [Queue](#) will always increase by a minimum value, regardless of the growth factor; a growth factor of 1.0 will not prevent the [Queue](#) from increasing in size.

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](#).

See

[Dequeue\(\)](#)[Dequeue\(\)](#)

Also

[Peek\(\)](#)[Peek\(\)](#)

# Queue.GetEnumerator Queue.GetEnumerator

## In this Article

Returns an enumerator that iterates through the [Queue](#).

```
public virtual System.Collections.IEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the [Queue](#).

## 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 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.

This method is an O(1) operation.

See

[IEnumerator](#)[IEnumerator](#)

Also

# Queue.IsSynchronized Queue.IsSynchronized

## In this Article

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

```
public virtual bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean](#)

`true` if access to the [Queue](#) is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration. Retrieving the value of this property is an O(1) operation.

```
Queue myCollection = new Queue();  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

The following example shows how to synchronize a [Queue](#), determine if a [Queue](#) is synchronized, and use a synchronized [Queue](#).

```
using System;  
using System.Collections;  
public class SamplesQueue {  
  
    public static void Main() {  
  
        // Creates and initializes a new Queue.  
        Queue myQ = new Queue();  
        myQ.Enqueue( "The" );  
        myQ.Enqueue( "quick" );  
        myQ.Enqueue( "brown" );  
        myQ.Enqueue( "fox" );  
  
        // Creates a synchronized wrapper around the Queue.  
        Queue mySyncdQ = Queue.Synchronized( myQ );  
  
        // Displays the synchronization status of both Queues.  
        Console.WriteLine( "myQ is {0}.", myQ.IsSynchronized ? "synchronized" : "not synchronized" );  
        Console.WriteLine( "mySyncdQ is {0}.", mySyncdQ.IsSynchronized ? "synchronized" : "not synchronized" );  
    }  
}  
/*  
This code produces the following output.  
  
myQ is not synchronized.  
mySyncdQ is synchronized.  
*/
```

## Remarks

To guarantee the thread safety of the [Queue](#), all operations must be done through the wrapper returned by the [Synchronized](#) method.

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 causes 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.

See

[SyncRoot](#)

Also

[Synchronized\(Queue\)](#)

# Queue.Peek Queue.Peek

## In this Article

Returns the object at the beginning of the [Queue](#) without removing it.

```
public virtual object Peek ();  
  
abstract member Peek : unit -> obj  
override this.Peek : unit -> obj
```

Returns

[Object Object](#)

The object at the beginning of the [Queue](#).

Exceptions

[InvalidOperationException InvalidOperationException](#)

The [Queue](#) is empty.

## Examples

The following example shows how to add elements to the [Queue](#), remove elements from the [Queue](#), or view the element at the beginning of the [Queue](#).

```

using System;
using System.Collections;
public class SamplesQueue {

    public static void Main() {

        // Creates and initializes a new Queue.
        Queue myQ = new Queue();
        myQ.Enqueue( "The" );
        myQ.Enqueue( "quick" );
        myQ.Enqueue( "brown" );
        myQ.Enqueue( "fox" );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Removes an element from the Queue.
        Console.WriteLine( "(Dequeue) {0}", myQ.Dequeue() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Removes another element from the Queue.
        Console.WriteLine( "(Dequeue) {0}", myQ.Dequeue() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );

        // Views the first element in the Queue but does not remove it.
        Console.WriteLine( "(Peek) {0}", myQ.Peek() );

        // Displays the Queue.
        Console.Write( "Queue values:" );
        PrintValues( myQ );
    }

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

/*
This code produces the following output.

Queue values: The quick brown fox
(Dequeue) The
Queue values: quick brown fox
(Dequeue) quick
Queue values: brown fox
(Peek) brown
Queue values: brown fox

*/

```

## Remarks

This method is similar to the [Dequeue](#) method, but [Peek](#) does not modify the [Queue](#).

[null](#) can be added to the [Queue](#) as a value. To distinguish between a null value and the end of the [Queue](#), check the

[Count](#) property or catch the [InvalidOperationException](#), which is thrown when the [Queue](#) is empty.

This method is an O(1) operation.

See

[Enqueue\(Object\)](#)

Also

[Dequeue\(\)](#)

[Dequeue\(\)](#)

# Queue Queue

## In this Article

## Overloads

<a href="#">Queue()</a>	Initializes a new instance of the <a href="#">Queue</a> class that is empty, has the default initial capacity, and uses the default growth factor.
<a href="#">Queue(ICollection)</a> <a href="#">Queue(ICollection)</a>	Initializes a new instance of the <a href="#">Queue</a> class that contains elements copied from the specified collection, has the same initial capacity as the number of elements copied, and uses the default growth factor.
<a href="#">Queue(Int32)</a> <a href="#">Queue(Int32)</a>	Initializes a new instance of the <a href="#">Queue</a> class that is empty, has the specified initial capacity, and uses the default growth factor.
<a href="#">Queue(Int32, Single)</a> <a href="#">Queue(Int32, Single)</a>	Initializes a new instance of the <a href="#">Queue</a> class that is empty, has the specified initial capacity, and uses the specified growth factor.

## Queue()

Initializes a new instance of the [Queue](#) class that is empty, has the default initial capacity, and uses the default growth factor.

```
public Queue ();
```

### Remarks

The capacity of a [Queue](#) is the number of elements the [Queue](#) can hold. As elements are added to a [Queue](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#).

The growth factor is the number by which the current capacity is multiplied when a greater capacity is required. The growth factor is determined when the [Queue](#) is constructed.

This constructor is an O(1) operation.

## Queue(ICollection) Queue(ICollection)

Initializes a new instance of the [Queue](#) class that contains elements copied from the specified collection, has the same initial capacity as the number of elements copied, and uses the default growth factor.

```
public Queue (System.Collections.ICollection col);  
new System.Collections.Queue : System.Collections.ICollection -> System.Collections.Queue
```

### Parameters

col

ICollection ICollection

The [ICollection](#) to copy elements from.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`col` is `null`.

Remarks

The capacity of a [Queue](#) is the number of elements the [Queue](#) can hold. As elements are added to a [Queue](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#).

The growth factor is the number by which the current capacity is multiplied when a greater capacity is required. The growth factor is determined when the [Queue](#) is constructed.

The elements are copied onto the [Queue](#) in the same order they are read by the [IEnumerator](#) of the [ICollection](#).

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

See

[ICollection](#) [Collection](#)

Also

## Queue(Int32) Queue(Int32)

Initializes a new instance of the [Queue](#) class that is empty, has the specified initial capacity, and uses the default growth factor.

```
public Queue (int capacity);
new System.Collections.Queue : int -> System.Collections.Queue
```

Parameters

capacity [Int32](#) [Int32](#)

The initial number of elements that the [Queue](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

Remarks

The capacity of a [Queue](#) is the number of elements the [Queue](#) can hold. As elements are added to a [Queue](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#).

The growth factor is the number by which the current capacity is multiplied when a greater capacity is required. The growth factor is determined when the [Queue](#) is constructed.

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](#).

This constructor is an  $O(n)$  operation, where `n` is `capacity`.

## Queue(Int32, Single) Queue(Int32, Single)

Initializes a new instance of the [Queue](#) class that is empty, has the specified initial capacity, and uses the specified growth factor.

```
public Queue (int capacity, float growFactor);  
new System.Collections.Queue : int * single -> System.Collections.Queue
```

## Parameters

capacity Int32 Int32

The initial number of elements that the [Queue](#) can contain.

growFactor Single Single

The factor by which the capacity of the [Queue](#) is expanded.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

-or-

`growFactor` is less than 1.0 or greater than 10.0.

## Remarks

The capacity of a [Queue](#) is the number of elements the [Queue](#) can hold. As elements are added to a [Queue](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#).

The growth factor is the number by which the current capacity is multiplied when a greater capacity is required. The growth factor is determined when the [Queue](#) is constructed. The capacity of the [Queue](#) will always increase by a minimum value, regardless of the growth factor; a growth factor of 1.0 will not prevent the [Queue](#) from increasing in size.

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](#).

This constructor is an  $O(n)$  operation, where  $n$  is `capacity`.

# Queue.Synchronized Queue.Synchronized

## In this Article

Returns a new [Queue](#) that wraps the original queue, and is thread safe.

```
public static System.Collections.Queue Synchronized (System.Collections.Queue queue);  
static member Synchronized : System.Collections.Queue -> System.Collections.Queue
```

### Parameters

queue	<a href="#">Queue</a> <a href="#">Queue</a>
-------	---

The [Queue](#) to synchronize.

### Returns

[Queue](#) [Queue](#)

A [Queue](#) wrapper that is synchronized (thread safe).

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`queue` is `null`.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration. This method is an O(1) operation.

```
Queue myCollection = new Queue();  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

The following example shows how to synchronize a [Queue](#), determine if a [Queue](#) is synchronized and use a synchronized [Queue](#).

```

using System;
using System.Collections;
public class SamplesQueue {

    public static void Main() {

        // Creates and initializes a new Queue.
        Queue myQ = new Queue();
        myQ.Enqueue( "The" );
        myQ.Enqueue( "quick" );
        myQ.Enqueue( "brown" );
        myQ.Enqueue( "fox" );

        // Creates a synchronized wrapper around the Queue.
        Queue mySyncdQ = Queue.Synchronized( myQ );

        // Displays the synchronization status of both Queues.
        Console.WriteLine( "myQ is {0}.", myQ.IsSynchronized ? "synchronized" : "not synchronized" );
        Console.WriteLine( "mySyncdQ is {0}.", mySyncdQ.IsSynchronized ? "synchronized" : "not synchronized" );
    }
}

/*
This code produces the following output.

myQ is not synchronized.
mySyncdQ is synchronized.
*/

```

## Remarks

The wrapper returned by this method locks the queue before an operation is performed so that it is performed in a thread-safe manner.

To guarantee the thread safety of the [Queue](#), all operations must be done through this wrapper only.

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 causes 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.

See

[IsSynchronized](#)

Also

[SyncRoot](#)

# Queue.SyncRoot Queue.SyncRoot

## In this Article

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

```
public virtual object SyncRoot { get; }  
member this.SyncRoot : obj
```

Returns

[Object Object](#)

An object that can be used to synchronize access to the [Queue](#).

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration. Retrieving the value of this property is an O(1) operation.

```
Queue myCollection = new Queue();  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

## Remarks

To create a synchronized version of the [Queue](#), use the [Synchronized](#) method. However, derived classes can provide their own synchronized version of the [Queue](#) using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) of the [Queue](#), not directly on the [Queue](#). This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [Queue](#) object.

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 causes 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.

See

[IsSynchronized](#)

Also

[Synchronized\(Queue\)](#)

[Synchronized\(Queue\)](#)

# Queue.ToArray Queue.ToArray

## In this Article

Copies the [Queue](#) elements to a new array.

```
public virtual object[] ToArray ();  
  
abstract member ToArray : unit -> obj[]  
override this.ToArray : unit -> obj[]
```

Returns

[Object](#)[]

A new array containing elements copied from the [Queue](#).

## Examples

The following example shows how to copy a [Queue](#) into a one-dimensional array.

```

using System;
using System.Collections;
public class SamplesQueue {

    public static void Main() {

        // Creates and initializes the source Queue.
        Queue mySourceQ = new Queue();
        mySourceQ.Enqueue( "three" );
        mySourceQ.Enqueue( "napping" );
        mySourceQ.Enqueue( "cats" );
        mySourceQ.Enqueue( "in" );
        mySourceQ.Enqueue( "the" );
        mySourceQ.Enqueue( "barn" );

        // Creates and initializes the one-dimensional target Array.
        Array myTargetArray=Array.CreateInstance( typeof(String), 15 );
        myTargetArray.SetValue( "The", 0 );
        myTargetArray.SetValue( "quick", 1 );
        myTargetArray.SetValue( "brown", 2 );
        myTargetArray.SetValue( "fox", 3 );
        myTargetArray.SetValue( "jumps", 4 );
        myTargetArray.SetValue( "over", 5 );
        myTargetArray.SetValue( "the", 6 );
        myTargetArray.SetValue( "lazy", 7 );
        myTargetArray.SetValue( "dog", 8 );

        // Displays the values of the target Array.
        Console.WriteLine( "The target Array contains the following (before and after copying):" );
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source Queue to the target Array, starting at index 6.
        mySourceQ.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source Queue to a new standard array.
        Object[] myStandardArray = mySourceQ.ToArray();

        // Displays the values of the new standard array.
        Console.WriteLine( "The new standard array contains the following:" );
        PrintValues( myStandardArray, ' ' );
    }

    public static void PrintValues( Array myArr, char mySeparator ) {
        foreach ( Object myObj in myArr ) {
            Console.Write( "{0}{1}", mySeparator, myObj );
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The target Array contains the following (before and after copying):
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over three napping cats in the barn
The new standard array contains the following:
three napping cats in the barn
*/

```

## Remarks

The [Queue](#) 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](#) to its end.

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

# Queue.TrimToSize Queue.TrimToSize

## In this Article

Sets the capacity to the actual number of elements in the [Queue](#).

```
public virtual void TrimToSize ();  
  
abstract member TrimToSize : unit -> unit  
override this.TrimToSize : unit -> unit
```

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [Queue](#) is read-only.

## Remarks

This method can be used to minimize a queue's memory overhead if no new elements will be added to the queue.

To reset a [Queue](#) to its initial state, call the [Clear](#) method before calling [TrimToSize](#). Trimming an empty [Queue](#) sets the capacity of the [Queue](#) to the default capacity.

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

See

[Clear\(\)](#)[Clear\(\)](#)

Also

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

# ReadOnlyCollectionBase ReadOnlyCollectionBase Class

Provides the `abstract` base class for a strongly typed non-generic read-only collection.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public abstract class ReadOnlyCollectionBase : System.Collections.ICollection

type ReadOnlyCollectionBase = class
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

`Object` `Object`

## Remarks

A `ReadOnlyCollectionBase` instance is always read-only. See `CollectionBase` for a modifiable version of this class.

### Important

We don't recommend that you use the `ReadOnlyCollectionBase` class for new development. Instead, we recommend that you use the generic `ReadOnlyCollection<T>` class. For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

## Constructors

```
ReadOnlyCollectionBase()
ReadOnlyCollectionBase()
```

Initializes a new instance of the `ReadOnlyCollectionBase` class.

## Properties

`Count`

`Count`

Gets the number of elements contained in the `ReadOnlyCollectionBase` instance.

`InnerList`

`InnerList`

Gets the list of elements contained in the `ReadOnlyCollectionBase` instance.

## Methods

```
GetEnumerator()
GetEnumerator()
```

Returns an enumerator that iterates through the `ReadOnlyCollectionBase` instance.

`ICollection.CopyTo(Array, Int32)`  
`ICollection.CopyTo(Array, Int32)`

Copies the entire [ReadOnlyCollectionBase](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

`ICollection.IsSynchronized`  
`ICollection.IsSynchronized`

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

`ICollection.SyncRoot`  
`ICollection.SyncRoot`

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

## 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.

This implementation does not provide a synchronized (thread safe) wrapper for a [ReadOnlyCollectionBase](#), but derived classes can create their own synchronized versions of the [ReadOnlyCollectionBase](#) using the [SyncRoot](#) property.

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 causes 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.

## See Also

[ArrayList](#) [ArrayList](#)

# ReadOnlyCollectionBase.Count

## In this Article

Gets the number of elements contained in the [ReadOnlyCollectionBase](#) instance.

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

```
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [ReadOnlyCollectionBase](#) instance.

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

## Examples

The following code example implements the [ReadOnlyCollectionBase](#) class.

```
using System;
using System.Collections;

public class ROCollection : ReadOnlyCollectionBase {
    public ROCollection( IList sourceList ) {
        InnerList.AddRange( sourceList );
    }

    public Object this[ int index ] {
        get {
            return( InnerList[index] );
        }
    }

    public int IndexOf( Object value ) {
        return( InnerList.IndexOf( value ) );
    }

    public bool Contains( Object value ) {
        return( InnerList.Contains( value ) );
    }
}

public class SamplesCollectionBase {
    public static void Main() {

        // Create an ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add( "red" );
        myAL.Add( "blue" );
        myAL.Add( "yellow" );
        myAL.Add( "green" );
        myAL.Add( "orange" );
        myAL.Add( "purple" );

        // Create a new ROCollection that contains the elements in myAL.
    }
}
```

```

ROCollection myCol = new ROCollection( myAL );

// Display the contents of the collection using foreach. This is the preferred method.
Console.WriteLine( "Contents of the collection (using foreach):" );
PrintValues1( myCol );

// Display the contents of the collection using the enumerator.
Console.WriteLine( "Contents of the collection (using enumerator):" );
PrintValues2( myCol );

// Display the contents of the collection using the Count property and the Item property.
Console.WriteLine( "Contents of the collection (using Count and Item):" );
PrintIndexAndValues( myCol );

// Search the collection with Contains and IndexOf.
Console.WriteLine( "Contains yellow: {0}", myCol.Contains( "yellow" ) );
Console.WriteLine( "orange is at index {0}.", myCol.IndexOf( "orange" ) );
Console.WriteLine();

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( ROCollection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( " [{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( ROCollection myCol ) {
    foreach ( Object obj in myCol )
        Console.WriteLine( " {0}", obj );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( ROCollection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( " {0}", myEnumerator.Current );
    Console.WriteLine();
}

}

/*
This code produces the following output.

Contents of the collection (using foreach):
red
blue
yellow
green
orange
purple

Contents of the collection (using enumerator):
red
blue
yellow
green
orange
purple

```

```
purple
```

```
Contents of the collection (using Count and Item):
```

```
[0]: red
[1]: blue
[2]: yellow
[3]: green
[4]: orange
[5]: purple
```

```
Contains yellow: True
```

```
orange is at index 4.
```

```
*/
```

# ReadOnlyCollectionBase.GetEnumerator ReadOnlyCollectionBase.GetEnumerator

## In this Article

Returns an enumerator that iterates through the [ReadOnlyCollectionBase](#) instance.

```
public virtual System.Collections.IEnumerator GetEnumerator ();  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the [ReadOnlyCollectionBase](#) instance.

## Examples

The following code example implements the [ReadOnlyCollectionBase](#) class.

```
using System;  
using System.Collections;  
  
public class ROCollection : ReadOnlyCollectionBase {  
  
    public ROCollection( IList sourceList ) {  
        InnerList.AddRange( sourceList );  
    }  
  
    public Object this[ int index ] {  
        get {  
            return( InnerList[index] );  
        }  
    }  
  
    public int IndexOf( Object value ) {  
        return( InnerList.IndexOf( value ) );  
    }  
  
    public bool Contains( Object value ) {  
        return( InnerList.Contains( value ) );  
    }  
  
}  
  
public class SamplesCollectionBase {  
  
    public static void Main() {  
  
        // Create an ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "red" );  
        myAL.Add( "blue" );  
        myAL.Add( "yellow" );  
        myAL.Add( "green" );  
        myAL.Add( "orange" );  
        myAL.Add( "purple" );  
  
        // Create a new ROCollection that contains the elements in myAL.  
        ROCollection myCol = new ROCollection( myAL );
```

```

// Display the contents of the collection using foreach. This is the preferred method.
Console.WriteLine( "Contents of the collection (using foreach):" );
PrintValues1( myCol );

// Display the contents of the collection using the enumerator.
Console.WriteLine( "Contents of the collection (using enumerator):" );
PrintValues2( myCol );

// Display the contents of the collection using the Count property and the Item property.
Console.WriteLine( "Contents of the collection (using Count and Item):" );
PrintIndexAndValues( myCol );

// Search the collection with Contains and IndexOf.
Console.WriteLine( "Contains yellow: {0}", myCol.Contains( "yellow" ) );
Console.WriteLine( "orange is at index {0}.", myCol.IndexOf( "orange" ) );
Console.WriteLine();

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( ROCollection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( " [{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( ROCollection myCol ) {
    foreach ( Object obj in myCol )
        Console.WriteLine( " {0}", obj );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( ROCollection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( " {0}", myEnumerator.Current );
    Console.WriteLine();
}

}

/*
This code produces the following output.

Contents of the collection (using foreach):
red
blue
yellow
green
orange
purple

Contents of the collection (using enumerator):
red
blue
yellow
green
orange
purple

```

```
Contents of the collection (using Count and Item):
```

```
[0]: red
[1]: blue
[2]: yellow
[3]: green
[4]: orange
[5]: purple
```

```
Contains yellow: True
orange is at index 4.
```

```
*/
```

## 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 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.

This method is an O(1) operation.

See

[IEnumerator](#)[IEnumerator](#)

Also

# ReadOnlyCollectionBase.ICollection.CopyTo

## In this Article

Copies the entire [ReadOnlyCollectionBase](#) to a compatible one-dimensional [Array](#), starting at the specified index of the target array.

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

## Parameters

array [Array](#)

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

index [Int32](#)

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

## Exceptions

[ArgumentNullException](#)

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

[ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#)

[array](#) is multidimensional.

-or-

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

[InvalidCastException](#)

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

## Remarks

The specified array must be of a compatible type.

This method uses [Array.Copy](#) to copy the elements.

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

# ReadOnlyCollectionBase.ICollection.IsSynchronized

## In this Article

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

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

Returns

[Boolean](#)

`true` if access to the [ReadOnlyCollectionBase](#) object is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) property during the entire enumeration.

```
// Get the ICollection interface from the ReadOnlyCollectionBase
// derived class.
ICollection myCollection = myReadOnlyCollection;
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

## Remarks

A [ReadOnlyCollectionBase](#) object is not synchronized. Derived classes can provide a synchronized version of the [ReadOnlyCollectionBase](#) class using the [SyncRoot](#) property.

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 causes 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.

# ReadOnlyCollectionBase.ICollection.SyncRoot

## In this Article

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

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

Returns

[Object](#)

An object that can be used to synchronize access to the [ReadOnlyCollectionBase](#) object.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) property during the entire enumeration.

```
// Get the ICollection interface from the ReadOnlyCollectionBase
// derived class.
ICollection myCollection = myReadOnlyCollection;
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

## Remarks

Derived classes can provide their own synchronized version of the [ReadOnlyCollectionBase](#) class using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) property of the [ReadOnlyCollectionBase](#) object, not directly on the [ReadOnlyCollectionBase](#) object. This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [ReadOnlyCollectionBase](#) object.

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 causes 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.

# ReadOnlyCollectionBase.InnerList ReadOnlyCollectionBase.InnerList

## In this Article

Gets the list of elements contained in the [ReadOnlyCollectionBase](#) instance.

```
protected System.Collections.ArrayList InnerList { get; }  
member this.InnerList : System.Collections.ArrayList
```

## Returns

[ArrayList](#) [ArrayList](#)

An [ArrayList](#) representing the [ReadOnlyCollectionBase](#) instance itself.

## Examples

The following code example implements the [ReadOnlyCollectionBase](#) class.

```
using System;  
using System.Collections;  
  
public class ROCollection : ReadOnlyCollectionBase {  
  
    public ROCollection( IList sourceList ) {  
        InnerList.AddRange( sourceList );  
    }  
  
    public Object this[ int index ] {  
        get {  
            return( InnerList[index] );  
        }  
    }  
  
    public int IndexOf( Object value ) {  
        return( InnerList.IndexOf( value ) );  
    }  
  
    public bool Contains( Object value ) {  
        return( InnerList.Contains( value ) );  
    }  
  
}  
  
public class SamplesCollectionBase {  
  
    public static void Main() {  
  
        // Create an ArrayList.  
        ArrayList myAL = new ArrayList();  
        myAL.Add( "red" );  
        myAL.Add( "blue" );  
        myAL.Add( "yellow" );  
        myAL.Add( "green" );  
        myAL.Add( "orange" );  
        myAL.Add( "purple" );  
  
        // Create a new ROCollection that contains the elements in myAL.  
        ROCollection myCol = new ROCollection( myAL );
```

```

// Display the contents of the collection using foreach. This is the preferred method.
Console.WriteLine( "Contents of the collection (using foreach):" );
PrintValues1( myCol );

// Display the contents of the collection using the enumerator.
Console.WriteLine( "Contents of the collection (using enumerator):" );
PrintValues2( myCol );

// Display the contents of the collection using the Count property and the Item property.
Console.WriteLine( "Contents of the collection (using Count and Item):" );
PrintIndexAndValues( myCol );

// Search the collection with Contains and IndexOf.
Console.WriteLine( "Contains yellow: {0}", myCol.Contains( "yellow" ) );
Console.WriteLine( "orange is at index {0}.", myCol.IndexOf( "orange" ) );
Console.WriteLine();

}

// Uses the Count property and the Item property.
public static void PrintIndexAndValues( ROCollection myCol ) {
    for ( int i = 0; i < myCol.Count; i++ )
        Console.WriteLine( " [{0}]: {1}", i, myCol[i] );
    Console.WriteLine();
}

// Uses the foreach statement which hides the complexity of the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues1( ROCollection myCol ) {
    foreach ( Object obj in myCol )
        Console.WriteLine( " {0}", obj );
    Console.WriteLine();
}

// Uses the enumerator.
// NOTE: The foreach statement is the preferred way of enumerating the contents of a collection.
public static void PrintValues2( ROCollection myCol ) {
    System.Collections.IEnumerator myEnumerator = myCol.GetEnumerator();
    while ( myEnumerator.MoveNext() )
        Console.WriteLine( " {0}", myEnumerator.Current );
    Console.WriteLine();
}
}

```

/\*
This code produces the following output.

Contents of the collection (using foreach):  
red  
blue  
yellow  
green  
orange  
purple

Contents of the collection (using enumerator):  
red  
blue  
yellow  
green  
orange  
purple

Contents of the collection (using Count and Item):

CONTENTS OF THE COLLECTION (USING COUNT AND ITEM).

```
[0]: red
[1]: blue
[2]: yellow
[3]: green
[4]: orange
[5]: purple
```

Contains yellow: True  
orange is at index 4.

```
*/
```

## Remarks

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

# ReadOnlyCollectionBase

## In this Article

Initializes a new instance of the [ReadOnlyCollectionBase](#) class.

```
protected ReadOnlyCollectionBase ();
```

## Remarks

This constructor is an O(1) operation.

# SortedList SortedList Class

Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public class SortedList : ICloneable, System.Collections.IDictionary

type SortedList = class
    interface IDictionary
    interface ICloneable
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

A [SortedList](#) element can be accessed by its key, like an element in any [IDictionary](#) implementation, or by its index, like an element in any [IList](#) implementation.

**Important**

We don't recommend that you use the [SortedList](#) class for new development. Instead, we recommend that you use the generic [System.Collections.Generic.SortedList<TKey,TValue>](#) class. For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

A [SortedList](#) object internally maintains two arrays to store the elements of the list; that is, one array for the keys and another array for the associated values. Each element is a key/value pair that can be accessed as a [DictionaryEntry](#) object. A key cannot be [null](#), but a value can be.

The capacity of a [SortedList](#) object is the number of elements the [SortedList](#) can hold. As elements are added to a [SortedList](#), the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling [TrimToSize](#) or by setting the [Capacity](#) property explicitly.

For very large [SortedList](#) objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the [enabled](#) attribute of the configuration element to [true](#) in the run-time environment.

The elements of a [SortedList](#) object are sorted by the keys either according to a specific [IComparer](#) implementation specified when the [SortedList](#) is created or according to the [IComparable](#) implementation provided by the keys themselves. In either case, a [SortedList](#) does not allow duplicate keys.

The index sequence is based on the sort sequence. When an element is added, it is inserted into [SortedList](#) in the correct sort order, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from the [SortedList](#) object.

Operations on a [SortedList](#) object tend to be slower than operations on a [Hashtable](#) object because of the sorting. However, the [SortedList](#) offers more flexibility by allowing access to the values either through the associated keys or through the indexes.

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

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 [SortedList](#) object is a key/value pair, the element type is not the type of the key or the type of the value. Rather, the element type is [DictionaryEntry](#). For example:

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

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

## Constructors

[SortedList\(\)](#)

[SortedList\(\)](#)

Initializes a new instance of the [SortedList](#) class that is empty, has the default initial capacity, and is sorted according to the [IComparable](#) interface implemented by each key added to the [SortedList](#) object.

[SortedList\(IComparer\)](#)

[SortedList\(IComparer\)](#)

Initializes a new instance of the [SortedList](#) class that is empty, has the default initial capacity, and is sorted according to the specified [IComparer](#) interface.

[SortedList\(IDictionary\)](#)

[SortedList\(IDictionary\)](#)

Initializes a new instance of the [SortedList](#) class that contains elements copied from the specified dictionary, has the same initial capacity as the number of elements copied, and is sorted according to the [IComparable](#) interface implemented by each key.

[SortedList\(Int32\)](#)

[SortedList\(Int32\)](#)

Initializes a new instance of the [SortedList](#) class that is empty, has the specified initial capacity, and is sorted according to the [IComparable](#) interface implemented by each key added to the [SortedList](#) object.

[SortedList\(IComparer, Int32\)](#)

[SortedList\(IComparer, Int32\)](#)

Initializes a new instance of the [SortedList](#) class that is empty, has the specified initial capacity, and is sorted according to the specified [IComparer](#) interface.

[SortedList\(IDictionary, IComparer\)](#)

[SortedList\(IDictionary, IComparer\)](#)

Initializes a new instance of the [SortedList](#) class that contains elements copied from the specified dictionary, has the same initial capacity as the number of elements copied, and is sorted according to the specified [IComparer](#)

interface.

## Properties

**Capacity**

**Capacity**

Gets or sets the capacity of a [SortedList](#) object.

**Count**

**Count**

Gets the number of elements contained in a [SortedList](#) object.

**IsFixedSize**

**IsFixedSize**

Gets a value indicating whether a [SortedList](#) object has a fixed size.

**IsReadOnly**

**IsReadOnly**

Gets a value indicating whether a [SortedList](#) object is read-only.

**IsSynchronized**

**IsSynchronized**

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

**Item[Object]**

**Item[Object]**

Gets and sets the value associated with a specific key in a [SortedList](#) object.

**Keys**

**Keys**

Gets the keys in a [SortedList](#) object.

**SyncRoot**

**SyncRoot**

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

Values

Values

Gets the values in a [SortedList](#) object.

## Methods

Add(Object, Object)

Add(Object, Object)

Adds an element with the specified key and value to a [SortedList](#) object.

Clear()

Clear()

Removes all elements from a [SortedList](#) object.

Clone()

Clone()

Creates a shallow copy of a [SortedList](#) object.

Contains(Object)

Contains(Object)

Determines whether a [SortedList](#) object contains a specific key.

ContainsKey(Object)

ContainsKey(Object)

Determines whether a [SortedList](#) object contains a specific key.

ContainsValue(Object)

ContainsValue(Object)

Determines whether a [SortedList](#) object contains a specific value.

CopyTo(Array, Int32)

CopyTo(Array, Int32)

Copies [SortedList](#) elements to a one-dimensional [Array](#) object, starting at the specified index in the array.

GetByIndex(Int32)

GetByIndex(Int32)

Gets the value at the specified index of a [SortedList](#) object.

```
GetEnumerator()  
GetEnumerator()
```

Returns an [IDictionaryEnumerator](#) object that iterates through a [SortedList](#) object.

```
GetKey(Int32)  
GetKey(Int32)
```

Gets the key at the specified index of a [SortedList](#) object.

```
GetKeyList()  
GetKeyList()
```

Gets the keys in a [SortedList](#) object.

```
GetValueList()  
GetValueList()
```

Gets the values in a [SortedList](#) object.

```
IndexOfKey(Object)  
IndexOfKey(Object)
```

Returns the zero-based index of the specified key in a [SortedList](#) object.

```
IndexOfValue(Object)  
IndexOfValue(Object)
```

Returns the zero-based index of the first occurrence of the specified value in a [SortedList](#) object.

```
Remove(Object)  
Remove(Object)
```

Removes the element with the specified key from a [SortedList](#) object.

```
RemoveAt(Int32)  
RemoveAt(Int32)
```

Removes the element at the specified index of a [SortedList](#) object.

```
SetByIndex(Int32, Object)  
SetByIndex(Int32, Object)
```

Replaces the value at a specific index in a [SortedList](#) object.

[Synchronized\(SortedList\)](#)

[Synchronized\(SortedList\)](#)

Returns a synchronized (thread-safe) wrapper for a [SortedList](#) object.

[TrimToSize\(\)](#)

[TrimToSize\(\)](#)

Sets the capacity to the actual number of elements in a [SortedList](#) object.

[IEnumerable.GetEnumerator\(\)](#)

[IEnumerable.GetEnumerator\(\)](#)

Returns an [IEnumerator](#) that iterates through the [SortedList](#).

## 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](#) object can support multiple readers concurrently, as long as the collection is not modified. To guarantee the thread safety of the [SortedList](#), all operations must be done through the wrapper returned by the [Synchronized\(SortedList\)](#) method.

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 causes 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.

## See Also

[IComparer](#) [IComparer](#)

[IDictionary](#) [IDictionary](#)

[IComparer](#) [IComparer](#)

# SortedList.Add SortedList.Add

## In this Article

Adds an element with the specified key and value to a [SortedList](#) object.

```
public virtual void Add (object key, object value);  
  
abstract member Add : obj * obj -> unit  
override this.Add : obj * obj -> unit
```

## Parameters

key Object Object

The key of the element to add.

value Object Object

The value of the element to add. The value can be `null`.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

[ArgumentException](#) [ArgumentException](#)

An element with the specified `key` already exists in the [SortedList](#) object.

-or-

The [SortedList](#) is set to use the [IComparable](#) interface, and `key` does not implement the [IComparable](#) interface.

[NotSupportedException](#) [NotSupportedException](#)

The [SortedList](#) is read-only.

-or-

The [SortedList](#) has a fixed size.

[OutOfMemoryException](#) [OutOfMemoryException](#)

There is not enough available memory to add the element to the [SortedList](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The comparer throws an exception.

## Examples

The following code example shows how to add elements to a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( "one", "The" );
        mySL.Add( "two", "quick" );
        mySL.Add( "three", "brown" );
        mySL.Add( "four", "fox" );

        // Displays the SortedList.
        Console.WriteLine( "The SortedList contains the following:" );
        PrintKeysAndValues( mySL );
    }

    public static void PrintKeysAndValues( SortedList myList ) {
        Console.WriteLine( " -KEY- -VALUE-" );
        for ( int i = 0; i < myList.Count; i++ ) {
            Console.WriteLine( " {0}: {1}", myList.GetKey(i), myList.GetByIndex(i) );
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The SortedList contains the following:
-KEY- -VALUE-
four: fox
one: The
three: brown
two: quick
*/

```

## Remarks

The insertion point is determined based on the comparer selected, either explicitly or by default, when the [SortedList](#) object was created.

If [Count](#) already equals [Capacity](#), the capacity of the [SortedList](#) object is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

You can also use the [Item\[Object\]](#) property to add new elements by setting the value of a key that does not exist in the [SortedList](#) object (for example, `myCollection["myNonexistentKey"] = myValue`). However, if the specified key already exists in the [SortedList](#), setting the [Item\[Object\]](#) property overwrites the old value. In contrast, the [Add](#) method does not modify existing elements.

The elements of a [SortedList](#) object are sorted by the keys either according to a specific [IComparer](#) implementation specified when the [SortedList](#) is created or according to the [IComparable](#) implementation provided by the keys themselves.

A key cannot be `null`, but a value can be.

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  $O(n)$ .

See

[Item\[Object\]](#)[Item\[Object\]](#)

Also

[IComparer](#)

[Capacity](#)

# SortedList.Capacity

## In this Article

Gets or sets the capacity of a [SortedList](#) object.

```
public virtual int Capacity { get; set; }  
member this.Capacity : int with get, set
```

Returns

[Int32](#)

The number of elements that the [SortedList](#) object can contain.

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

The value assigned is less than the current number of elements in the [SortedList](#) object.

[OutOfMemoryException](#) [OutOfMemoryException](#)

There is not enough memory available on the system.

## Remarks

[Capacity](#) is the number of elements that the [SortedList](#) object can store. [Count](#) is the number of elements that are actually in the [SortedList](#).

[Capacity](#) is always greater than or equal to [Count](#). If [Count](#) exceeds [Capacity](#) while adding elements, the capacity is automatically increased by reallocating the internal array before copying the old elements and adding the new elements.

The capacity can be decreased by calling [TrimToSize](#) 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.

See

[TrimToSize\(\)](#)

Also

# SortedList.Clear

## In this Article

Removes all elements from a [SortedList](#) object.

```
public virtual void Clear ();  
  
abstract member Clear : unit -> unit  
override this.Clear : unit -> unit
```

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [SortedList](#) object is read-only.

-or-

The [SortedList](#) has a fixed size.

## Examples

The following code example shows how to trim the unused portions of a [SortedList](#) object and how to clear the values of the [SortedList](#).

```
using System;  
using System.Collections;  
public class SamplesSortedList {  
  
    public static void Main() {  
  
        // Creates and initializes a new SortedList.  
        SortedList mySL = new SortedList();  
        mySL.Add( "one", "The" );  
        mySL.Add( "two", "quick" );  
        mySL.Add( "three", "brown" );  
        mySL.Add( "four", "fox" );  
        mySL.Add( "five", "jumps" );  
  
        // Displays the count, capacity and values of the SortedList.  
        Console.WriteLine( "Initially," );  
        Console.WriteLine( "    Count      : {0}", mySL.Count );  
        Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );  
        Console.WriteLine( "    Values:" );  
        PrintKeysAndValues( mySL );  
  
        // Trims the SortedList.  
        mySL.TrimToSize();  
  
        // Displays the count, capacity and values of the SortedList.  
        Console.WriteLine( "After TrimToSize," );  
        Console.WriteLine( "    Count      : {0}", mySL.Count );  
        Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );  
        Console.WriteLine( "    Values:" );  
        PrintKeysAndValues( mySL );  
  
        // Clears the SortedList.  
        mySL.Clear();  
  
        // Displays the count, capacity and values of the SortedList.  
        Console.WriteLine( "After Clear," );  
        Console.WriteLine( "    Count      : {0}", mySL.Count );  
        Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );
```

```

        Console.WriteLine( "    Values:" );
        PrintKeysAndValues( mySL );

        // Trims the SortedList again.
        mySL.TrimToSize();

        // Displays the count, capacity and values of the SortedList.
        Console.WriteLine( "After the second TrimToSize," );
        Console.WriteLine( "    Count      : {0}", mySL.Count );
        Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );
        Console.WriteLine( "    Values:" );
        PrintKeysAndValues( mySL );
    }

public static void PrintKeysAndValues( SortedList myList ) {
    Console.WriteLine( "    -KEY-    -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( "    {0}:    {1}", myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}
/*
This code produces the following output.

Initially,
Count      : 5
Capacity   : 16
Values:
-KEY-    -VALUE-
five:    jumps
four:    fox
one:    The
three:   brown
two:    quick

After TrimToSize,
Count      : 5
Capacity   : 5
Values:
-KEY-    -VALUE-
five:    jumps
four:    fox
one:    The
three:   brown
two:    quick

After Clear,
Count      : 0
Capacity   : 16
Values:
-KEY-    -VALUE-

After the second TrimToSize,
Count      : 0
Capacity   : 16
Values:
-KEY-    -VALUE-
*/

```

## 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](#) object, call [TrimToSize](#) or set the [Capacity](#) property directly. Trimming an empty [SortedList](#) sets the capacity of the [SortedList](#) to the default capacity.

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

See

[TrimToSize\(\)](#)

Also

[Capacity](#)

[Count](#)

# SortedList.Clone SortedList.Clone

## In this Article

Creates a shallow copy of a [SortedList](#) object.

```
public virtual object Clone ();  
  
abstract member Clone : unit -> obj  
override this.Clone : unit -> obj
```

Returns

[Object Object](#)

A shallow copy of the [SortedList](#) object.

## Remarks

A shallow copy of a collection copies only the elements of the collection, whether they are reference types or value types, but does not copy the objects that the references refer to. The references in the new collection point to the same objects that the references in the original collection point to.

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](#).

See

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

Also

# SortedList.Contains

## In this Article

Determines whether a [SortedList](#) object contains a specific key.

```
public virtual bool Contains (object key);  
  
abstract member Contains : obj -> bool  
override this.Contains : obj -> bool
```

## Parameters

**key** [Object](#) [Object](#)

The key to locate in the [SortedList](#) object.

## Returns

[Boolean](#) [Boolean](#)

`true` if the [SortedList](#) object contains an element with the specified `key`; otherwise, `false`.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

[InvalidOperationException](#) [InvalidOperationException](#)

The comparer throws an exception.

## Examples

The following code example shows how to determine whether a [SortedList](#) object contains a specific element.

```

using System;
using System.Collections;

public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 2, "two" );
        mySL.Add( 4, "four" );
        mySL.Add( 1, "one" );
        mySL.Add( 3, "three" );
        mySL.Add( 0, "zero" );

        // Displays the values of the SortedList.
        Console.WriteLine( "The SortedList contains the following values:" );
        PrintIndexAndKeysAndValues( mySL );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"{0}\" is {1}.", myKey, mySL.ContainsKey( myKey ) ? "in the
        SortedList" : "NOT in the SortedList" );
        myKey = 6;
        Console.WriteLine( "The key \"{0}\" is {1}.", myKey, mySL.ContainsKey( myKey ) ? "in the
        SortedList" : "NOT in the SortedList" );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"{0}\" is {1}.", myValue, mySL.ContainsValue( myValue ) ? "in
        the SortedList" : "NOT in the SortedList" );
        myValue = "nine";
        Console.WriteLine( "The value \"{0}\" is {1}.", myValue, mySL.ContainsValue( myValue ) ? "in
        the SortedList" : "NOT in the SortedList" );
    }
}

```

```

public static void PrintIndexAndKeysAndValues( SortedList myList ) {
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( " [{0}]: {1} {2}", i, myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}

/*
This code produces the following output.

```

The SortedList contains the following values:

-INDEX-	-KEY-	-VALUE-
[0]:	0	zero
[1]:	1	one
[2]:	2	two
[3]:	3	three
[4]:	4	four

The key "2" is in the SortedList.

The key "6" is NOT in the SortedList.

The value "three" is in the SortedList.

The value "nine" is NOT in the SortedList.

\*/

## Remarks

The elements of a [SortedList](#) object are sorted by the keys either according to a specific [IComparer](#) implementation specified when the [SortedList](#) is created or according to the [IComparable](#) implementation provided by the keys themselves.

[Contains](#) implements [IDictionary.Contains](#). It behaves exactly as [ContainsKey](#).

This method uses a binary search algorithm; therefore, this method is an  $O(\log n)$  operation, where  $n$  is [Count](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[ContainsKey\(Object\)](#)

Also

[IndexOfKey\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.ContainsKey SortedList.ContainsKey

## In this Article

Determines whether a [SortedList](#) object contains a specific key.

```
public virtual bool ContainsKey (object key);  
  
abstract member ContainsKey : obj -> bool  
override this.ContainsKey : obj -> bool
```

## Parameters

**key** [Object](#) [Object](#)

The key to locate in the [SortedList](#) object.

## Returns

[Boolean](#) [Boolean](#)

`true` if the [SortedList](#) object contains an element with the specified `key`; otherwise, `false`.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

[InvalidOperationException](#) [InvalidOperationException](#)

The comparer throws an exception.

## Examples

The following code example shows how to determine whether a [SortedList](#) object contains a specific element.

```

using System;
using System.Collections;

public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 2, "two" );
        mySL.Add( 4, "four" );
        mySL.Add( 1, "one" );
        mySL.Add( 3, "three" );
        mySL.Add( 0, "zero" );

        // Displays the values of the SortedList.
        Console.WriteLine( "The SortedList contains the following values:" );
        PrintIndexAndKeysAndValues( mySL );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"{0}\" is {1}.", myKey, mySL.ContainsKey( myKey ) ? "in the
        SortedList" : "NOT in the SortedList" );
        myKey = 6;
        Console.WriteLine( "The key \"{0}\" is {1}.", myKey, mySL.ContainsKey( myKey ) ? "in the
        SortedList" : "NOT in the SortedList" );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"{0}\" is {1}.", myValue, mySL.ContainsValue( myValue ) ? "in
        the SortedList" : "NOT in the SortedList" );
        myValue = "nine";
        Console.WriteLine( "The value \"{0}\" is {1}.", myValue, mySL.ContainsValue( myValue ) ? "in
        the SortedList" : "NOT in the SortedList" );
    }
}

```

```

public static void PrintIndexAndKeysAndValues( SortedList myList ) {
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( " [{0}]: {1} {2}", i, myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}

/*
This code produces the following output.

```

The SortedList contains the following values:

-INDEX-	-KEY-	-VALUE-
[0]:	0	zero
[1]:	1	one
[2]:	2	two
[3]:	3	three
[4]:	4	four

The key "2" is in the SortedList.

The key "6" is NOT in the SortedList.

The value "three" is in the SortedList.

The value "nine" is NOT in the SortedList.

\*/

## Remarks

The elements of a [SortedList](#) object are sorted by the keys either according to a specific [IComparer](#) implementation specified when the [SortedList](#) is created or according to the [IComparable](#) implementation provided by the keys themselves.

This method behaves exactly as the [Contains](#) method.

This method uses a binary search algorithm; therefore, this method is an  $O(\log n)$  operation, where  $n$  is [Count](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[Contains\(Object\)](#)

Also

[ContainsValue\(Object\)](#)

[IndexOfKey\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.ContainsValue SortedList.ContainsValue

## In this Article

Determines whether a [SortedList](#) object contains a specific value.

```
public virtual bool ContainsValue (object value);  
  
abstract member ContainsValue : obj -> bool  
override this.ContainsValue : obj -> bool
```

## Parameters

**value** [Object](#) [Object](#)

The value to locate in the [SortedList](#) object. The value can be `null`.

## Returns

[Boolean](#) [Boolean](#)

`true` if the [SortedList](#) object contains an element with the specified `value`; otherwise, `false`.

## Examples

The following code example shows how to determine whether a [SortedList](#) object contains a specific element.

```

using System;
using System.Collections;

public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 2, "two" );
        mySL.Add( 4, "four" );
        mySL.Add( 1, "one" );
        mySL.Add( 3, "three" );
        mySL.Add( 0, "zero" );

        // Displays the values of the SortedList.
        Console.WriteLine( "The SortedList contains the following values:" );
        PrintIndexAndKeysAndValues( mySL );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"{0}\" is {1}.", myKey, mySL.ContainsKey( myKey ) ? "in the
        SortedList" : "NOT in the SortedList" );
        myKey = 6;
        Console.WriteLine( "The key \"{0}\" is {1}.", myKey, mySL.ContainsKey( myKey ) ? "in the
        SortedList" : "NOT in the SortedList" );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"{0}\" is {1}.", myValue, mySL.ContainsValue( myValue ) ? "in
        the SortedList" : "NOT in the SortedList" );
        myValue = "nine";
        Console.WriteLine( "The value \"{0}\" is {1}.", myValue, mySL.ContainsValue( myValue ) ? "in
        the SortedList" : "NOT in the SortedList" );
    }
}

```

```

public static void PrintIndexAndKeysAndValues( SortedList myList ) {
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( " [{0}]: {1} {2}", i, myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}

/*
This code produces the following output.

```

The SortedList contains the following values:

-INDEX-	-KEY-	-VALUE-
[0]:	0	zero
[1]:	1	one
[2]:	2	two
[3]:	3	three
[4]:	4	four

The key "2" is in the SortedList.

The key "6" is NOT in the SortedList.

The value "three" is in the SortedList.

The value "nine" is NOT in the SortedList.

\*/

## Remarks

The values of the elements of the [SortedList](#) object are compared to the specified value using the [Equals](#) method.

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](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[ContainsKey\(Object\)](#)

Also

[IndexOfValue\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.CopyTo SortedList.CopyTo

## In this Article

Copies [SortedList](#) elements to a one-dimensional [Array](#) object, starting at the specified index in the array.

```
public virtual void CopyTo (Array array, int arrayIndex);  
  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

## Parameters

array [Array](#) [Array](#)

The one-dimensional [Array](#) object that is the destination of the [DictionaryEntry](#) objects copied from [SortedList](#). The [Array](#) must have zero-based indexing.

arrayIndex [Int32](#) [Int32](#)

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

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[arrayIndex](#) is less than zero.

[ArgumentException](#) [ArgumentException](#)

[array](#) is multidimensional.

-or-

The number of elements in the source [SortedList](#) object is greater than the available space from [arrayIndex](#) to the end of the destination [array](#).

[InvalidCastException](#) [InvalidCastException](#)

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

## Examples

The following code example shows how to copy the values in a [SortedList](#) object into a one-dimensional [Array](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes the source SortedList.
        SortedList mySourceList = new SortedList();
        mySourceList.Add( 2, "cats" );
        mySourceList.Add( 3, "in" );
        mySourceList.Add( 1, "napping" );
        mySourceList.Add( 4, "the" );
        mySourceList.Add( 0, "three" );
        mySourceList.Add( 5, "barn" );

        // Creates and initializes the one-dimensional target Array.
        String[] tempArray = new String[] { "The", "quick", "brown", "fox", "jumps", "over", "the",
        "lazy", "dog" };
        DictionaryEntry[] myTargetArray = new DictionaryEntry[15];
        int i = 0;
        foreach ( String s in tempArray ) {
            myTargetArray[i].Key = i;
            myTargetArray[i].Value = s;
            i++;
        }

        // Displays the values of the target Array.
        Console.WriteLine( "The target Array contains the following (before and after copying):" );
        PrintValues( myTargetArray, ' ' );

        // Copies the entire source SortedList to the target SortedList, starting at index 6.
        mySourceList.CopyTo( myTargetArray, 6 );

        // Displays the values of the target Array.
        PrintValues( myTargetArray, ' ' );
    }

    public static void PrintValues( DictionaryEntry[] myArr, char mySeparator ) {
        for ( int i = 0; i < myArr.Length; i++ )
            Console.Write( "{0}{1}", mySeparator, myArr[i].Value );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The target Array contains the following (before and after copying):
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over three napping cats in the barn
*/

```

## Remarks

The key/value pairs are copied to the [Array](#) object in the same order in which the enumerator iterates through the [SortedList](#) object.

To copy only the keys in the [SortedList](#), use [SortedList.Keys.CopyTo](#).

To copy only the values in the [SortedList](#), use [SortedList.Values.CopyTo](#).

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

See

Also

[Array](#)[Array](#)

[GetEnumerator\(\)](#)[GetEnumerator\(\)](#)

# SortedList.Count

## In this Article

Gets the number of elements contained in a [SortedList](#) object.

```
public virtual int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [SortedList](#) object.

## Remarks

Each element is a key/value pair that can be accessed as a [DictionaryEntry](#) object.

[Capacity](#) is the number of elements that the [SortedList](#) object can store. [Count](#) is the number of elements that are actually in the [SortedList](#).

[Capacity](#) is always greater than or equal to [Count](#). If [Count](#) exceeds [Capacity](#) while adding elements, the capacity is automatically increased by 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.

See

[Capacity](#)

Also

# SortedList.GetByIndex SortedList.GetByIndex

## In this Article

Gets the value at the specified index of a [SortedList](#) object.

```
public virtual object GetByIndex (int index);  
  
abstract member GetByIndex : int -> obj  
override this.GetByIndex : int -> obj
```

## Parameters

index [Int32](#) [Int32](#)

The zero-based index of the value to get.

## Returns

[Object Object](#)

The value at the specified index of the [SortedList](#) object.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is outside the range of valid indexes for the [SortedList](#) object.

## Examples

The following code example shows how to get one or all the keys or values in a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 1.3, "fox" );
        mySL.Add( 1.4, "jumps" );
        mySL.Add( 1.5, "over" );
        mySL.Add( 1.2, "brown" );
        mySL.Add( 1.1, "quick" );
        mySL.Add( 1.0, "The" );
        mySL.Add( 1.6, "the" );
        mySL.Add( 1.8, "dog" );
        mySL.Add( 1.7, "lazy" );

        // Gets the key and the value based on the index.
        int myIndex=3;
        Console.WriteLine( "The key at index {0} is {1}.", myIndex, mySL.GetKey( myIndex ) );
        Console.WriteLine( "The value at index {0} is {1}.", myIndex, mySL.GetByIndex( myIndex ) );

        // Gets the list of keys and the list of values.
        IList myKeyList = mySL.GetKeyList();
        IList myValueList = mySL.GetValueList();

        // Prints the keys in the first column and the values in the second column.
        Console.WriteLine( " -KEY- -VALUE-" );
        for ( int i = 0; i < mySL.Count; i++ )
            Console.WriteLine( " {0} {1}", myKeyList[i], myValueList[i] );
    }
}

/*
This code produces the following output.

The key at index 3 is 1.3.
The value at index 3 is fox.
-KEY- -VALUE-
1 The
1.1 quick
1.2 brown
1.3 fox
1.4 jumps
1.5 over
1.6 the
1.7 lazy
1.8 dog
*/

```

## Remarks

The index sequence is based on the sort sequence. When an element is added, it is inserted into [SortedList](#) in the correct sort order, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from the [SortedList](#) object.

This method is an O(1) operation.

See

Also

[IndexOfKey\(Object\)](#)[IndexOfKey\(Object\)](#)

[IndexOfValue\(Object\)](#)[IndexOfValue\(Object\)](#)

# SortedList.GetEnumerator SortedList.GetEnumerator

## In this Article

Returns an [IDictionaryEnumerator](#) object that iterates through a [SortedList](#) object.

```
public virtual System.Collections.IDictionaryEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IDictionaryEnumerator  
override this.GetEnumerator : unit -> System.Collections.IDictionaryEnumerator
```

Returns

[IDictionaryEnumerator](#) [IDictionaryEnumerator](#)

An [IDictionaryEnumerator](#) object for the [SortedList](#) object.

## 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 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.

This method is an O(1) operation.

See

[IDictionaryEnumerator](#)[IDictionaryEnumerator](#)

Also

[IEnumerator](#)[IEnumerator](#)

# SortedList.GetKey SortedList.GetKey

## In this Article

Gets the key at the specified index of a [SortedList](#) object.

```
public virtual object GetKey (int index);  
  
abstract member GetKey : int -> obj  
override this.GetKey : int -> obj
```

## Parameters

index [Int32](#) [Int32](#)

The zero-based index of the key to get.

## Returns

[Object Object](#)

The key at the specified index of the [SortedList](#) object.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is outside the range of valid indexes for the [SortedList](#) object.

## Examples

The following code example shows how to get one or all the keys or values in a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 1.3, "fox" );
        mySL.Add( 1.4, "jumps" );
        mySL.Add( 1.5, "over" );
        mySL.Add( 1.2, "brown" );
        mySL.Add( 1.1, "quick" );
        mySL.Add( 1.0, "The" );
        mySL.Add( 1.6, "the" );
        mySL.Add( 1.8, "dog" );
        mySL.Add( 1.7, "lazy" );

        // Gets the key and the value based on the index.
        int myIndex=3;
        Console.WriteLine( "The key at index {0} is {1}.", myIndex, mySL.GetKey( myIndex ) );
        Console.WriteLine( "The value at index {0} is {1}.", myIndex, mySL.GetByIndex( myIndex ) );

        // Gets the list of keys and the list of values.
        IList myKeyList = mySL.GetKeyList();
        IList myValueList = mySL.GetValueList();

        // Prints the keys in the first column and the values in the second column.
        Console.WriteLine( " -KEY- -VALUE-" );
        for ( int i = 0; i < mySL.Count; i++ )
            Console.WriteLine( " {0} {1}", myKeyList[i], myValueList[i] );
    }
}

/*
This code produces the following output.

The key at index 3 is 1.3.
The value at index 3 is fox.
-KEY- -VALUE-
1 The
1.1 quick
1.2 brown
1.3 fox
1.4 jumps
1.5 over
1.6 the
1.7 lazy
1.8 dog
*/

```

## Remarks

The index sequence is based on the sort sequence. When an element is added, it is inserted into [SortedList](#) in the correct sort order, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from the [SortedList](#) object.

This method is an O(1) operation.

# SortedList.GetKeyList SortedList.GetKeyList

## In this Article

Gets the keys in a [SortedList](#) object.

```
public virtual System.Collections.IList GetKeyList ();  
  
abstract member GetKeyList : unit -> System.Collections.IList  
override this.GetKeyList : unit -> System.Collections.IList
```

## Returns

[IList](#) [IList](#)

An [IList](#) object containing the keys in the [SortedList](#) object.

## Examples

The following code example shows how to get one or all the keys or values in a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 1.3, "fox" );
        mySL.Add( 1.4, "jumps" );
        mySL.Add( 1.5, "over" );
        mySL.Add( 1.2, "brown" );
        mySL.Add( 1.1, "quick" );
        mySL.Add( 1.0, "The" );
        mySL.Add( 1.6, "the" );
        mySL.Add( 1.8, "dog" );
        mySL.Add( 1.7, "lazy" );

        // Gets the key and the value based on the index.
        int myIndex=3;
        Console.WriteLine( "The key at index {0} is {1}.", myIndex, mySL.GetKey( myIndex ) );
        Console.WriteLine( "The value at index {0} is {1}.", myIndex, mySL.GetByIndex( myIndex ) );

        // Gets the list of keys and the list of values.
        IList myKeyList = mySL.GetKeyList();
        IList myValueList = mySL.GetValueList();

        // Prints the keys in the first column and the values in the second column.
        Console.WriteLine( " -KEY- -VALUE-" );
        for ( int i = 0; i < mySL.Count; i++ )
            Console.WriteLine( " {0} {1}", myKeyList[i], myValueList[i] );
    }
}

/*
This code produces the following output.

The key at index 3 is 1.3.
The value at index 3 is fox.
-KEY- -VALUE-
1 The
1.1 quick
1.2 brown
1.3 fox
1.4 jumps
1.5 over
1.6 the
1.7 lazy
1.8 dog
*/

```

## Remarks

The returned [IList](#) object is a read-only view of the keys of the [SortedList](#) object. Modifications made to the underlying [SortedList](#) are immediately reflected in the [IList](#).

The elements of the returned [IList](#) are sorted in the same order as the keys of the [SortedList](#).

This method is similar to the [Keys](#) property, but returns an [IList](#) object instead of an [ICollection](#) object.

This method is an O(1) operation.

See

[IList](#)  
[IList](#)

Also

[GetValueList\(\)](#)  
[GetValueList\(\)](#)



# SortedList.GetValueList SortedList.GetValueList

## In this Article

Gets the values in a [SortedList](#) object.

```
public virtual System.Collections.IList GetValueList ();  
  
abstract member GetValueList : unit -> System.Collections.IList  
override this.GetValueList : unit -> System.Collections.IList
```

## Returns

[IList](#) [IList](#)

An [IList](#) object containing the values in the [SortedList](#) object.

## Examples

The following code example shows how to get one or all the keys or values in a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 1.3, "fox" );
        mySL.Add( 1.4, "jumps" );
        mySL.Add( 1.5, "over" );
        mySL.Add( 1.2, "brown" );
        mySL.Add( 1.1, "quick" );
        mySL.Add( 1.0, "The" );
        mySL.Add( 1.6, "the" );
        mySL.Add( 1.8, "dog" );
        mySL.Add( 1.7, "lazy" );

        // Gets the key and the value based on the index.
        int myIndex=3;
        Console.WriteLine( "The key at index {0} is {1}.", myIndex, mySL.GetKey( myIndex ) );
        Console.WriteLine( "The value at index {0} is {1}.", myIndex, mySL.GetByIndex( myIndex ) );

        // Gets the list of keys and the list of values.
        IList myKeyList = mySL.GetKeyList();
        IList myValueList = mySL.GetValueList();

        // Prints the keys in the first column and the values in the second column.
        Console.WriteLine( " -KEY- -VALUE-" );
        for ( int i = 0; i < mySL.Count; i++ )
            Console.WriteLine( " {0} {1}", myKeyList[i], myValueList[i] );
    }
}

/*
This code produces the following output.

The key at index 3 is 1.3.
The value at index 3 is fox.
-KEY- -VALUE-
1 The
1.1 quick
1.2 brown
1.3 fox
1.4 jumps
1.5 over
1.6 the
1.7 lazy
1.8 dog
*/

```

## Remarks

The returned [IList](#) object is a read-only view of the values of the [SortedList](#) object. Modifications made to the underlying [SortedList](#) are immediately reflected in the [IList](#).

The elements of the returned [IList](#) are sorted in the same order as the values of the [SortedList](#).

This method is similar to the [Values](#) property, but returns an [IList](#) object instead of an [ICollection](#) object.

This method is an O(1) operation.

See

Also

[IList](#)  
[GetKeyList\(\)](#)

[GetKeyList\(\)](#)



# SortedList.IEnumerable.GetEnumerator

## In this Article

Returns an [IEnumerator](#) that iterates through the [SortedList](#).

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

Returns

[IEnumerator](#)

An [IEnumerator](#) for the [SortedList](#).

## Remarks

[Visual Basic, C#]

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, calling [Current](#) throws an exception. 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`, calling [Current](#) throws an exception. 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](#). If the collection is modified between [MoveNext](#) and [Current](#), [Current](#) returns the element that it is set to, even if the enumerator is already invalidated.

The enumerator does not have exclusive access to the collection; therefore, 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 causes 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.

This method is an O(1) operation.

See

Also

[IDictionaryEnumerator](#)  
[IEnumerator](#)

# SortedList.IndexOfKey

## In this Article

Returns the zero-based index of the specified key in a [SortedList](#) object.

```
public virtual int IndexOfKey (object key);  
  
abstract member IndexOfKey : obj -> int  
override this.IndexOfKey : obj -> int
```

### Parameters

**key** [Object](#) [Object](#)

The key to locate in the [SortedList](#) object.

### Returns

[Int32](#) [Int32](#)

The zero-based index of the `key` parameter, if `key` is found in the [SortedList](#) object; otherwise, -1.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

[InvalidOperationException](#) [InvalidOperationException](#)

The comparer throws an exception.

## Examples

The following code example shows how to determine the index of a key or a value in a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 1, "one" );
        mySL.Add( 3, "three" );
        mySL.Add( 2, "two" );
        mySL.Add( 4, "four" );
        mySL.Add( 0, "zero" );

        // Displays the values of the SortedList.
        Console.WriteLine( "The SortedList contains the following values:" );
        PrintIndexAndKeysAndValues( mySL );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"{0}\" is at index {1}.", myKey, mySL.IndexOfKey( myKey ) );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"{0}\" is at index {1}.", myValue, mySL.IndexOfValue( myValue ) );
    }
}

public static void PrintIndexAndKeysAndValues( SortedList myList ) {
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( " [{0}]: {1} {2}", i, myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}
/*
This code produces the following output.

```

The SortedList contains the following values:

-INDEX-	-KEY-	-VALUE-
[0]:	0	zero
[1]:	1	one
[2]:	2	two
[3]:	3	three
[4]:	4	four

The key "2" is at index 2.  
 The value "three" is at index 3.  
 \*/

## Remarks

The elements of a [SortedList](#) object are sorted by the keys either according to a specific [IComparer](#) implementation specified when the [SortedList](#) is created, or according to the [IComparable](#) implementation provided by the keys themselves.

The index sequence is based on the sort sequence. When an element is added, it is inserted into [SortedList](#) in the correct sort order, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from the [SortedList](#).

This method uses a binary search algorithm; therefore, this method is an  $O(\log n)$  operation, where  $n$  is [Count](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on [item](#) to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the [item](#) parameter on the objects in the collection.

See

[ContainsKey\(Object\)](#)

Also

[IndexOfValue\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.IndexOfValue SortedList.IndexOfValue

## In this Article

Returns the zero-based index of the first occurrence of the specified value in a [SortedList](#) object.

```
public virtual int IndexOfValue (object value);  
  
abstract member IndexOfValue : obj -> int  
override this.IndexOfValue : obj -> int
```

## Parameters

**value** [Object](#) [Object](#)

The value to locate in the [SortedList](#) object. The value can be `null`.

## Returns

[Int32](#) [Int32](#)

The zero-based index of the first occurrence of the `value` parameter, if `value` is found in the [SortedList](#) object; otherwise, -1.

## Examples

The following code example shows how to determine the index of a key or a value in a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 1, "one" );
        mySL.Add( 3, "three" );
        mySL.Add( 2, "two" );
        mySL.Add( 4, "four" );
        mySL.Add( 0, "zero" );

        // Displays the values of the SortedList.
        Console.WriteLine( "The SortedList contains the following values:" );
        PrintIndexAndKeysAndValues( mySL );

        // Searches for a specific key.
        int myKey = 2;
        Console.WriteLine( "The key \"{0}\" is at index {1}.", myKey, mySL.IndexOfKey( myKey ) );

        // Searches for a specific value.
        String myValue = "three";
        Console.WriteLine( "The value \"{0}\" is at index {1}.", myValue, mySL.IndexOfValue( myValue ) );
    }
}

public static void PrintIndexAndKeysAndValues( SortedList myList ) {
    Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( " [{0}]: {1} {2}", i, myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}
/*
This code produces the following output.

```

The SortedList contains the following values:

-INDEX-	-KEY-	-VALUE-
[0]:	0	zero
[1]:	1	one
[2]:	2	two
[3]:	3	three
[4]:	4	four

The key "2" is at index 2.  
 The value "three" is at index 3.  
 \*/

## Remarks

The index sequence is based on the sort sequence. When an element is added, it is inserted into [SortedList](#) in the correct sort order, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from the [SortedList](#) object.

The values of the elements of the [SortedList](#) are compared to the specified value using the [Equals](#) method.

This method uses a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

Starting with the .NET Framework 2.0, this method uses the collection's objects' [Equals](#) and [CompareTo](#) methods on `item` to determine whether item exists. In the earlier versions of the .NET Framework, this determination was made by using the [Equals](#) and [CompareTo](#) methods of the `item` parameter on the objects in the collection.

See

[ContainsValue\(Object\)](#)[ContainsValue\(Object\)](#)

Also

[IndexOfKey\(Object\)](#)[IndexOfKey\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.IsFixedSize SortedList.IsFixedSize

## In this Article

Gets a value indicating whether a [SortedList](#) object has a fixed size.

```
public virtual bool IsFixedSize { get; }  
member this.IsFixedSize : bool
```

Returns

[Boolean](#)

`true` if the [SortedList](#) object has a fixed size; otherwise, `false`. The default is `false`.

## Remarks

A collection with a fixed size does not allow the addition or removal of elements after the collection is created, but does allow 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.

# SortedList.IsReadOnly SortedList.IsReadOnly

## In this Article

Gets a value indicating whether a [SortedList](#) object is read-only.

```
public virtual bool IsReadOnly { get; }  
member this.IsReadOnly : bool
```

Returns

[Boolean](#)

`true` if the [SortedList](#) object is read-only; otherwise, `false`. The default is `false`.

## 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.

# SortedList.IsSynchronized SortedList.IsSynchronized

## In this Article

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

```
public virtual bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean](#)

`true` if access to the [SortedList](#) object is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Examples

The following code example shows how to lock a collection using the [SyncRoot](#) property during the entire enumeration.

```
SortedList myCollection = new SortedList();  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

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

The following code example shows how to synchronize a [SortedList](#) object, determine whether a [SortedList](#) is synchronized, and use a synchronized [SortedList](#).

```

using System;
using System.Collections;
public class SamplesSortedList  {

    public static void Main()  {

        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 2, "two" );
        mySL.Add( 3, "three" );
        mySL.Add( 1, "one" );
        mySL.Add( 0, "zero" );
        mySL.Add( 4, "four" );

        // Creates a synchronized wrapper around the SortedList.
        SortedList mySyncdSL = SortedList.Synchronized( mySL );

        // Displays the synchronization status of both SortedLists.
        Console.WriteLine( "mySL is {0}.". , mySL.IsSynchronized ? "synchronized" : "not synchronized"
);
        Console.WriteLine( "mySyncdSL is {0}.". , mySyncdSL.IsSynchronized ? "synchronized" : "not
synchronized" );
    }
}
/*
This code produces the following output.

mySL is not synchronized.
mySyncdSL is synchronized.
*/

```

## Remarks

To guarantee the thread safety of a [SortedList](#) object, all operations must be done through the wrapper returned by the [Synchronized](#) method.

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 causes 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.

See

[SyncRoot](#)

Also

[Synchronized\(SortedList\)](#)

# SortedList.Item[Object] SortedList.Item[Object]

## In this Article

Gets and sets the value associated with a specific key in a [SortedList](#) object.

```
public virtual object this[object key] { get; set; }  
member this.Item(obj) : obj with get, set
```

### Parameters

key [Object](#) [Object](#)

The key associated with the value to get or set.

### Returns

[Object](#) [Object](#)

The value associated with the `key` parameter in the [SortedList](#) object, if `key` is found; otherwise, `null`.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`key` is `null`.

[NotSupportedException](#) [NotSupportedException](#)

The property is set and the [SortedList](#) object is read-only.

-or-

The property is set, `key` does not exist in the collection, and the [SortedList](#) has a fixed size.

[OutOfMemoryException](#) [OutOfMemoryException](#)

There is not enough available memory to add the element to the [SortedList](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The comparer throws an exception.

## Remarks

You can use the [Item\[Object\]](#) property to access a specific element in a collection by specifying the following syntax:

`myCollection[key]`.

You can also use this property to add new elements by setting the value of a key that does not exist in the [SortedList](#) object (for example, `myCollection["myNonexistentKey"] = myValue`). However, if the specified key already exists in the [SortedList](#), setting the [Item\[Object\]](#) 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. To distinguish between `null` that is returned because the specified key is not found and `null` that is returned because the value of the specified key is `null`, use the [Contains](#) method or the [ContainsKey](#) method to determine if the key exists in the list.

The elements of a [SortedList](#) are sorted by the keys either according to a specific [IComparer](#) implementation specified when the [SortedList](#) is created or according to the [IComparable](#) implementation provided by the keys themselves.

The C# language uses the keyword to define the indexers instead of implementing the [Keys](#) property. Visual Basic

implements [Item\[Object\]](#) 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](#). 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  $O(n)$ .

See

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

Also

[Contains\(Object\)](#)  
[Contains\(Object\)](#)

[ContainsKey\(Object\)](#)  
[ContainsKey\(Object\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.Keys SortedList.Keys

## In this Article

Gets the keys in a [SortedList](#) object.

```
public virtual System.Collections.ICollection Keys { get; }  
member this.Keys : System.Collections.ICollection
```

Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) object containing the keys in the [SortedList](#) object.

## Remarks

The [ICollection](#) object is a read-only view of the keys of the [SortedList](#) object. Modifications made to the underlying [SortedList](#) are immediately reflected in the [ICollection](#).

The elements of the [ICollection](#) are sorted in the same order as the keys of the [SortedList](#).

This property is similar to the [GetKeyList](#) method, but returns an [ICollection](#) object instead of an [IList](#) object.

This method is an O(1) operation.

See

[ICollection](#)  
[ValueCollection](#)

Also

[GetKeyList\(\)](#)  
[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.Remove

## In this Article

Removes the element with the specified key from a [SortedList](#) object.

```
public virtual void Remove (object key);  
  
abstract member Remove : obj -> unit  
override this.Remove : obj -> unit
```

## Parameters

key	<a href="#">Object</a>
-----	------------------------

The key of the element to remove.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`key` is `null`.

[NotSupportedException](#) [NotSupportedException](#)

The [SortedList](#) object is read-only.

-or-

The [SortedList](#) has a fixed size.

## Examples

The following code example shows how to remove elements from a [SortedList](#) object.

```
using System;  
using System.Collections;  
public class SamplesSortedList {  
  
    public static void Main() {  
  
        // Creates and initializes a new SortedList.  
        SortedList mySL = new SortedList();  
        mySL.Add( "3c", "dog" );  
        mySL.Add( "2c", "over" );  
        mySL.Add( "1c", "brown" );  
        mySL.Add( "1a", "The" );  
        mySL.Add( "1b", "quick" );  
        mySL.Add( "3a", "the" );  
        mySL.Add( "3b", "lazy" );  
        mySL.Add( "2a", "fox" );  
        mySL.Add( "2b", "jumps" );  
  
        // Displays the SortedList.  
        Console.WriteLine( "The SortedList initially contains the following:" );  
        PrintKeysAndValues( mySL );  
  
        // Removes the element with the key "3b".  
        mySL.Remove( "3b" );  
  
        // Displays the current state of the SortedList.  
        Console.WriteLine( "After removing \"lazy\":" );  
        PrintKeysAndValues( mySL );  
    }  
}  
  
private static void PrintKeysAndValues( SortedList sl ) {  
    foreach ( object key in sl.Keys ) {  
        Console.WriteLine( "Key: " + key + " Value: " + sl[key] );  
    }  
}
```

```

// Removes the element at index 5.
mySL.RemoveAt( 5 );

// Displays the current state of the SortedList.
Console.WriteLine( "After removing the element at index 5:" );
PrintKeysAndValues( mySL );
}

public static void PrintKeysAndValues( SortedList myList ) {
    Console.WriteLine( " -KEY- -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( " {0}: {1}", myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}
/*
This code produces the following output.

```

The SortedList initially contains the following:

-KEY-	-VALUE-
1a:	The
1b:	quick
1c:	brown
2a:	fox
2b:	jumps
2c:	over
3a:	the
3b:	lazy
3c:	dog

After removing "lazy":

-KEY-	-VALUE-
1a:	The
1b:	quick
1c:	brown
2a:	fox
2b:	jumps
2c:	over
3a:	the
3c:	dog

After removing the element at index 5:

-KEY-	-VALUE-
1a:	The
1b:	quick
1c:	brown
2a:	fox
2b:	jumps
3a:	the
3c:	dog

\*/

## Remarks

If the [SortedList](#) object does not contain an element with the specified key, the [SortedList](#) remains unchanged. No exception is thrown.

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.

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

See

[RemoveAt\(Int32\)](#)

Also

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.RemoveAt

## In this Article

Removes the element at the specified index of a [SortedList](#) object.

```
public virtual void RemoveAt (int index);  
  
abstract member RemoveAt : int -> unit  
override this.RemoveAt : int -> unit
```

## Parameters

index	<a href="#">Int32</a>
-------	-----------------------

The zero-based index of the element to remove.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is outside the range of valid indexes for the [SortedList](#) object.

[NotSupportedException](#) [NotSupportedException](#)

The [SortedList](#) is read-only.

-or-

The [SortedList](#) has a fixed size.

## Examples

The following code example shows how to remove elements from a [SortedList](#) object.

```
using System;  
using System.Collections;  
public class SamplesSortedList {  
  
    public static void Main() {  
  
        // Creates and initializes a new SortedList.  
        SortedList mySL = new SortedList();  
        mySL.Add( "3c", "dog" );  
        mySL.Add( "2c", "over" );  
        mySL.Add( "1c", "brown" );  
        mySL.Add( "1a", "The" );  
        mySL.Add( "1b", "quick" );  
        mySL.Add( "3a", "the" );  
        mySL.Add( "3b", "lazy" );  
        mySL.Add( "2a", "fox" );  
        mySL.Add( "2b", "jumps" );  
  
        // Displays the SortedList.  
        Console.WriteLine( "The SortedList initially contains the following:" );  
        PrintKeysAndValues( mySL );  
  
        // Removes the element with the key "3b".  
        mySL.Remove( "3b" );  
  
        // Displays the current state of the SortedList.  
        Console.WriteLine( "After removing \"lazy\":" );  
        PrintKeysAndValues( mySL );  
    }  
}  
  
private static void PrintKeysAndValues( SortedList sl ) {  
    foreach (DictionaryEntry de in sl) {  
        Console.WriteLine( de.Key + " : " + de.Value );  
    }  
}
```

```

// Removes the element at index 5.
mySL.RemoveAt( 5 );

// Displays the current state of the SortedList.
Console.WriteLine( "After removing the element at index 5:" );
PrintKeysAndValues( mySL );
}

public static void PrintKeysAndValues( SortedList myList ) {
    Console.WriteLine( " -KEY- -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( " {0}: {1}", myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}
/*

```

This code produces the following output.

The SortedList initially contains the following:

-KEY-	-VALUE-
1a:	The
1b:	quick
1c:	brown
2a:	fox
2b:	jumps
2c:	over
3a:	the
3b:	lazy
3c:	dog

After removing "lazy":

-KEY-	-VALUE-
1a:	The
1b:	quick
1c:	brown
2a:	fox
2b:	jumps
2c:	over
3a:	the
3c:	dog

After removing the element at index 5:

-KEY-	-VALUE-
1a:	The
1b:	quick
1c:	brown
2a:	fox
2b:	jumps
3a:	the
3c:	dog

\*/

## Remarks

The index sequence is based on the sort sequence. When an element is added, it is inserted into [SortedList](#) in the correct sort order, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from the [SortedList](#) object.

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.

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

See

[Remove\(Object\)](#)[Remove\(Object\)](#)

Also

# SortedList.SetByIndex SortedList.SetByIndex

## In this Article

Replaces the value at a specific index in a [SortedList](#) object.

```
public virtual void SetByIndex (int index, object value);  
  
abstract member SetByIndex : int * obj -> unit  
override this.SetByIndex : int * obj -> unit
```

## Parameters

index Int32 Int32

The zero-based index at which to save `value`.

value Object Object

The [Object](#) to save into the [SortedList](#) object. The value can be `null`.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`index` is outside the range of valid indexes for the [SortedList](#) object.

## Examples

The following code example shows how to replace the value of an existing element in a [SortedList](#) object.

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 2, "two" );
        mySL.Add( 3, "three" );
        mySL.Add( 1, "one" );
        mySL.Add( 0, "zero" );
        mySL.Add( 4, "four" );

        // Displays the values of the SortedList.
        Console.WriteLine( "The SortedList contains the following values:" );
        PrintIndexAndKeysAndValues( mySL );

        // Replaces the values at index 3 and index 4.
        mySL.SetByIndex( 3, "III" );
        mySL.SetByIndex( 4, "IV" );

        // Displays the updated values of the SortedList.
        Console.WriteLine( "After replacing the value at index 3 and index 4," );
        PrintIndexAndKeysAndValues( mySL );
    }

    public static void PrintIndexAndKeysAndValues( SortedList myList ) {
        Console.WriteLine( " -INDEX- -KEY- -VALUE-" );
        for ( int i = 0; i < myList.Count; i++ ) {
            Console.WriteLine( " [{0}]: {1} {2}", i, myList.GetKey(i), myList.GetByIndex(i) );
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

The SortedList contains the following values:
-INDEX- -KEY- -VALUE-
[0]: 0 zero
[1]: 1 one
[2]: 2 two
[3]: 3 three
[4]: 4 four

After replacing the value at index 3 and index 4,
-INDEX- -KEY- -VALUE-
[0]: 0 zero
[1]: 1 one
[2]: 2 two
[3]: 3 III
[4]: 4 IV
*/

```

## Remarks

The index sequence is based on the sort sequence. When an element is added, it is inserted into [SortedList](#) in the correct sort order, and the indexing adjusts accordingly. When an element is removed, the indexing also adjusts accordingly. Therefore, the index of a specific key/value pair might change as elements are added or removed from the [SortedList](#) object.

This method is an O(1) operation.

See

Also

[IndexOfKey\(Object\)](#)

[IndexOfValue\(Object\)](#)

# SortedList SortedList

## In this Article

## Overloads

SortedList()	Initializes a new instance of the <a href="#">SortedList</a> class that is empty, has the default initial capacity, and is sorted according to the <a href="#">IComparable</a> interface implemented by each key added to the <a href="#">SortedList</a> object.
SortedList(IComparer) SortedList(IComparer)	Initializes a new instance of the <a href="#">SortedList</a> class that is empty, has the default initial capacity, and is sorted according to the specified <a href="#">IComparer</a> interface.
SortedList(IDictionary) SortedList(IDictionary)	Initializes a new instance of the <a href="#">SortedList</a> class that contains elements copied from the specified dictionary, has the same initial capacity as the number of elements copied, and is sorted according to the <a href="#">IComparable</a> interface implemented by each key.
SortedList(Int32) SortedList(Int32)	Initializes a new instance of the <a href="#">SortedList</a> class that is empty, has the specified initial capacity, and is sorted according to the <a href="#">IComparable</a> interface implemented by each key added to the <a href="#">SortedList</a> object.
SortedList(IComparer, Int32) SortedList(IComparer, Int32)	Initializes a new instance of the <a href="#">SortedList</a> class that is empty, has the specified initial capacity, and is sorted according to the specified <a href="#">IComparer</a> interface.
SortedList(IDictionary, IComparer) SortedList(IDictionary, IComparer)	Initializes a new instance of the <a href="#">SortedList</a> class that contains elements copied from the specified dictionary, has the same initial capacity as the number of elements copied, and is sorted according to the specified <a href="#">IComparer</a> interface.

## SortedList()

Initializes a new instance of the [SortedList](#) class that is empty, has the default initial capacity, and is sorted according to the [IComparable](#) interface implemented by each key added to the [SortedList](#) object.

```
public SortedList();
```

## Examples

The following code example creates collections using different [SortedList](#) constructors and demonstrates the differences in the behavior of the collections.

```
using System;
using System.Collections;
using System.Globalization;
```

```
public class SamplesSortedList
{
    public static void Main()
    {
        // Create a SortedList using the default comparer.
        SortedList mySL1 = new SortedList();
        Console.WriteLine("mySL1 (default):");
        mySL1.Add("FIRST", "Hello");
        mySL1.Add("SECOND", "World");
        mySL1.Add("THIRD", "!");
        try
        {
            mySL1.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL1);

        // Create a SortedList using the specified case-insensitive comparer.
        SortedList mySL2 = new SortedList(new CaseInsensitiveComparer());
        Console.WriteLine("mySL2 (case-insensitive comparer):");
        mySL2.Add("FIRST", "Hello");
        mySL2.Add("SECOND", "World");
        mySL2.Add("THIRD", "!");
        try
        {
            mySL2.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL2);

        // Create a SortedList using the specified CaseInsensitiveComparer,
        // which is based on the Turkish culture (tr-TR), where "I" is not
        // the uppercase version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        SortedList mySL3 = new SortedList(new CaseInsensitiveComparer(myCul));
        Console.WriteLine(
            "mySL3 (case-insensitive comparer, Turkish culture):");

        mySL3.Add("FIRST", "Hello");
        mySL3.Add("SECOND", "World");
        mySL3.Add("THIRD", "!");
        try
        {
            mySL3.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL3);

        // Create a SortedList using the
        // StringComparer.InvariantCultureIgnoreCase value.
        SortedList mySL4 = new SortedList(
            StringComparer.InvariantCultureIgnoreCase);

        Console.WriteLine("mySL4 (InvariantCultureIgnoreCase):");
    }
}
```

```

        Console.WriteLine("mySL4 (InvariantCultureIgnoreCase). ");
        mySL4.Add("FIRST", "Hello");
        mySL4.Add("SECOND", "World");
        mySL4.Add("THIRD", "!");
        try
        {
            mySL4.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL4);

    }

    public static void PrintKeysAndValues(SortedList myList)
    {
        Console.WriteLine("      -KEY-      -VALUE-");
        for (int i = 0; i < myList.Count; i++)
        {
            Console.WriteLine("      {0,-6}: {1}",
                myList.GetKey(i), myList.GetByIndex(i));
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

mySL1 (default):
      -KEY-      -VALUE-
      first : Ola!
      FIRST : Hello
      SECOND: World
      THIRD : !

mySL2 (case-insensitive comparer):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added: 'first'
   at System.Collections.SortedList.Add(Object key, Object value)
   at SamplesSortedList.Main()
      -KEY-      -VALUE-
      FIRST : Hello
      SECOND: World
      THIRD : !

mySL3 (case-insensitive comparer, Turkish culture):
      -KEY-      -VALUE-
      FIRST : Hello
      first : Ola!
      SECOND: World
      THIRD : !

mySL4 (InvariantCultureIgnoreCase):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added: 'first'
   at System.Collections.SortedList.Add(Object key, Object value)
   at SamplesSortedList.Main()
      -KEY-      -VALUE-
      FIRST : Hello
      SECOND: World
      THIRD : !

```

```
*/
```

## Remarks

Each key must implement the [IComparable](#) interface to be capable of comparisons with every other key in the [SortedList](#) object. The elements are sorted according to the [IComparable](#) implementation of each key added to the [SortedList](#).

The capacity of a [SortedList](#) object is the number of elements that the [SortedList](#) can hold. As elements are added to a [SortedList](#), 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](#) object.

This constructor is an O(1) operation.

See

[Capacity](#)

Also

[Performing Culture-Insensitive String Operations in Collections](#)

## SortedList(IComparer) SortedList(IComparer)

Initializes a new instance of the [SortedList](#) class that is empty, has the default initial capacity, and is sorted according to the specified [IComparer](#) interface.

```
public SortedList (System.Collections.IComparer comparer);  
new System.Collections.SortedList : System.Collections.IComparer -> System.Collections.SortedList
```

### Parameters

comparer

[IComparer](#)

The [IComparer](#) implementation to use when comparing keys.

-or-

`null` to use the [IComparable](#) implementation of each key.

### Examples

The following code example creates collections using different [SortedList](#) constructors and demonstrates the differences in the behavior of the collections.

```
using System;  
using System.Collections;  
using System.Globalization;  
  
public class SamplesSortedList  
{  
  
    public static void Main()  
    {  
  
        // Create a SortedList using the default comparer.  
        SortedList mySL1 = new SortedList();  
        Console.WriteLine("mySL1 (default):");  
        mySL1.Add("FIRST", "Hello");  
        mySL1.Add("SECOND", "World");  
        mySL1.Add("THIRD", "!");  
        try
```

```

{
    mySL1.Add("first", "Ola!");
}
catch (ArgumentException e)
{
    Console.WriteLine(e);
}
PrintKeysAndValues(mySL1);

// Create a SortedList using the specified case-insensitive comparer.
SortedList mySL2 = new SortedList(new CaseInsensitiveComparer());
Console.WriteLine("mySL2 (case-insensitive comparer):");
mySL2.Add("FIRST", "Hello");
mySL2.Add("SECOND", "World");
mySL2.Add("THIRD", "!");
try
{
    mySL2.Add("first", "Ola!");
}
catch (ArgumentException e)
{
    Console.WriteLine(e);
}
PrintKeysAndValues(mySL2);

// Create a SortedList using the specified CaseInsensitiveComparer,
// which is based on the Turkish culture (tr-TR), where "I" is not
// the uppercase version of "i".
CultureInfo myCul = new CultureInfo("tr-TR");
SortedList mySL3 = new SortedList(new CaseInsensitiveComparer(myCul));
Console.WriteLine(
    "mySL3 (case-insensitive comparer, Turkish culture):");

mySL3.Add("FIRST", "Hello");
mySL3.Add("SECOND", "World");
mySL3.Add("THIRD", "!");
try
{
    mySL3.Add("first", "Ola!");
}
catch (ArgumentException e)
{
    Console.WriteLine(e);
}
PrintKeysAndValues(mySL3);

// Create a SortedList using the
// StringComparer.InvariantCultureIgnoreCase value.
SortedList mySL4 = new SortedList(
    StringComparer.InvariantCultureIgnoreCase);

Console.WriteLine("mySL4 (InvariantCultureIgnoreCase):");
mySL4.Add("FIRST", "Hello");
mySL4.Add("SECOND", "World");
mySL4.Add("THIRD", "!");
try
{
    mySL4.Add("first", "Ola!");
}
catch (ArgumentException e)
{
    Console.WriteLine(e);
}
PrintKeysAndValues(mySL4);

```

```

}

public static void PrintKeysAndValues(SortedList myList)
{
    Console.WriteLine("      -KEY-      -VALUE-");
    for (int i = 0; i < myList.Count; i++)
    {
        Console.WriteLine("      {0,-6}: {1}",
            myList.GetKey(i), myList.GetByIndex(i));
    }
    Console.WriteLine();
}

/*
This code produces the following output.
Results vary depending on the system's culture settings.

mySL1 (default):
-KEY-      -VALUE-
first : Ola!
FIRST : Hello
SECOND: World
THIRD : !

mySL2 (case-insensitive comparer):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
at System.Collections.SortedList.Add(Object key, Object value)
at SamplesSortedList.Main()
-KEY-      -VALUE-
FIRST : Hello
SECOND: World
THIRD : !

mySL3 (case-insensitive comparer, Turkish culture):
-KEY-      -VALUE-
FIRST : Hello
first : Ola!
SECOND: World
THIRD : !

mySL4 (InvariantCultureIgnoreCase):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
at System.Collections.SortedList.Add(Object key, Object value)
at SamplesSortedList.Main()
-KEY-      -VALUE-
FIRST : Hello
SECOND: World
THIRD : !

*/

```

## Remarks

The elements are sorted according to the specified [IComparer](#) implementation. If the `comparer` parameter is `null`, the [IComparable](#) implementation of each key is used; therefore, each key must implement the [IComparable](#) interface to be capable of comparisons with every other key in the [SortedList](#) object.

The capacity of a [SortedList](#) object is the number of elements that the [SortedList](#) can hold. As elements are added to a [SortedList](#), 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](#) object.

This constructor is an O(1) operation.

See

[IComparer](#)

Also

[Capacity](#)

[Performing Culture-Insensitive String Operations in Collections](#)

## SortedList(IDictionary) SortedList(IDictionary)

Initializes a new instance of the [SortedList](#) class that contains elements copied from the specified dictionary, has the same initial capacity as the number of elements copied, and is sorted according to the [IComparable](#) interface implemented by each key.

```
public SortedList (System.Collections.IDictionary d);  
new System.Collections.SortedList : System.Collections.IDictionary -> System.Collections.SortedList
```

Parameters

d [IDictionary](#) [IDictionary](#)

The [IDictionary](#) implementation to copy to a new [SortedList](#) object.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

d is [null](#).

[InvalidOperationException](#) [InvalidOperationException](#)

One or more elements in [d](#) do not implement the [IComparable](#) interface.

Examples

The following code example creates collections using different [SortedList](#) constructors and demonstrates the differences in the behavior of the collections.

```
using System;  
using System.Collections;  
using System.Globalization;  
  
public class SamplesSortedList  
{  
  
    public static void Main()  
    {  
  
        // Create the dictionary.  
        Hashtable myHT = new Hashtable();  
        myHT.Add("FIRST", "Hello");  
        myHT.Add("SECOND", "World");  
        myHT.Add("THIRD", "!");  
  
        // Create a SortedList using the default comparer.  
        SortedList mySL1 = new SortedList(myHT);  
        Console.WriteLine("mySL1 (default):");  
        try  
        {  
            mySL1.Add("first", "Ola!");  
        }
```

```

        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL1);

        // Create a SortedList using the specified case-insensitive comparer.
        SortedList mySL2 = new SortedList(myHT, new CaseInsensitiveComparer());
        Console.WriteLine("mySL2 (case-insensitive comparer):");
        try
        {
            mySL2.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL2);

        // Create a SortedList using the specified CaseInsensitiveComparer,
        // which is based on the Turkish culture (tr-TR), where "I" is not
        // the uppercase version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        SortedList mySL3 = new SortedList(myHT, new CaseInsensitiveComparer(myCul));
        Console.WriteLine("mySL3 (case-insensitive comparer, Turkish culture):");
        try
        {
            mySL3.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL3);

        // Create a SortedList using the
        // StringComparer.InvariantCultureIgnoreCase value.
        SortedList mySL4 = new SortedList(
            myHT, StringComparer.InvariantCultureIgnoreCase);

        Console.WriteLine("mySL4 (InvariantCultureIgnoreCase):");
        try
        {
            mySL4.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL4);

    }

    public static void PrintKeysAndValues(SortedList myList)
    {
        Console.WriteLine("      -KEY-      -VALUE-");
        for (int i = 0; i < myList.Count; i++)
        {
            Console.WriteLine("      {0,-6}: {1}",
                myList.GetKey(i), myList.GetByIndex(i));
        }
        Console.WriteLine();
    }
}

```

```

/*
This code produces the following output. Results vary depending on the system's culture settings.

mySL1 (default):
    -KEY-    -VALUE-
    first : Ola!
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL2 (case-insensitive comparer):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-    -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL3 (case-insensitive comparer, Turkish culture):
    -KEY-    -VALUE-
    FIRST : Hello
    first : Ola!
    SECOND: World
    THIRD : !

mySL4 (InvariantCultureIgnoreCase):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-    -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

*/

```

## Remarks

Each key must implement the [IComparable](#) interface to be capable of comparisons with every other key in the [SortedList](#) object. The elements are sorted according to the [IComparable](#) implementation of each key added to the [SortedList](#).

A [Hashtable](#) object is an example of an [IDictionary](#) implementation that can be passed to this constructor. The new [SortedList](#) object contains a copy of the keys and values stored in the [Hashtable](#).

The capacity of a [SortedList](#) object is the number of elements that the [SortedList](#) can hold. As elements are added to a [SortedList](#), 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](#) object.

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

See

[IDictionary](#)

Also

[Capacity](#)

[Performing Culture-Insensitive String Operations in Collections](#)

## **SortedList(Int32) SortedList(Int32)**

Initializes a new instance of the [SortedList](#) class that is empty, has the specified initial capacity, and is sorted according to the [IComparable](#) interface implemented by each key added to the [SortedList](#) object.

```
public SortedList (int initialCapacity);
new System.Collections.SortedList : int -> System.Collections.SortedList
```

#### Parameters

initialCapacity Int32 Int32

The initial number of elements that the [SortedList](#) object can contain.

#### Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`initialCapacity` is less than zero.

[OutOfMemoryException](#) [OutOfMemoryException](#)

There is not enough available memory to create a [SortedList](#) object with the specified `initialCapacity`.

#### Examples

The following code example creates collections using different [SortedList](#) constructors and demonstrates the differences in the behavior of the collections.

```
using System;
using System.Collections;
using System.Globalization;

public class SamplesSortedList
{
    public static void Main()
    {
        // Create a SortedList using the default comparer.
        SortedList mySL1 = new SortedList( 3 );
        Console.WriteLine("mySL1 (default):");
        mySL1.Add("FIRST", "Hello");
        mySL1.Add("SECOND", "World");
        mySL1.Add("THIRD", "!");
        try
        {
            mySL1.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL1);

        // Create a SortedList using the specified case-insensitive comparer.
        SortedList mySL2 = new SortedList(new CaseInsensitiveComparer(), 3);
        Console.WriteLine("mySL2 (case-insensitive comparer):");
        mySL2.Add("FIRST", "Hello");
        mySL2.Add("SECOND", "World");
        mySL2.Add("THIRD", "!");
        try
        {
            mySL2.Add("first", "Ola!");
        }
```

```

        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL2);

        // Create a SortedList using the specified CaseInsensitiveComparer,
        // which is based on the Turkish culture (tr-TR), where "I" is not
        // the uppercase version of "i".
        CultureInfo myCul = new CultureInfo("tr-TR");
        SortedList mySL3 =
            new SortedList(new CaseInsensitiveComparer(myCul), 3);

        Console.WriteLine(
            "mySL3 (case-insensitive comparer, Turkish culture):");

        mySL3.Add("FIRST", "Hello");
        mySL3.Add("SECOND", "World");
        mySL3.Add("THIRD", "!");
        try
        {
            mySL3.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL3);

        // Create a SortedList using the
        // StringComparer.InvariantCultureIgnoreCase value.
        SortedList mySL4 = new SortedList(
            StringComparer.InvariantCultureIgnoreCase, 3);

        Console.WriteLine("mySL4 (InvariantCultureIgnoreCase):");
        mySL4.Add("FIRST", "Hello");
        mySL4.Add("SECOND", "World");
        mySL4.Add("THIRD", "!");
        try
        {
            mySL4.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL4);

    }

    public static void PrintKeysAndValues(SortedList myList)
    {
        Console.WriteLine("      -KEY-      -VALUE-");
        for (int i = 0; i < myList.Count; i++)
        {
            Console.WriteLine("      {0,-6}: {1}",
                myList.GetKey(i), myList.GetByIndex(i));
        }
        Console.WriteLine();
    }
}

/*
This code produces the following output.

```

Results vary depending on the system's culture settings.

```
mySL1 (default):
    -KEY-      -VALUE-
    first : Ola!
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL2 (case-insensitive comparer):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-      -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL3 (case-insensitive comparer, Turkish culture):
    -KEY-      -VALUE-
    FIRST : Hello
    first : Ola!
    SECOND: World
    THIRD : !

mySL4 (InvariantCultureIgnoreCase):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-      -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

*/
```

## Remarks

Each key must implement the [IComparable](#) interface to be capable of comparisons with every other key in the [SortedList](#) object. The elements are sorted according to the [IComparable](#) implementation of each key added to the [SortedList](#).

The capacity of a [SortedList](#) object is the number of elements that the [SortedList](#) can hold. As elements are added to a [SortedList](#), 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](#) object.

This constructor is an  $O(n)$  operation, where  $n$  is [initialCapacity](#).

See

[CapacityCapacity](#)

Also

[Performing Culture-Insensitive String Operations in Collections](#)

## **SortedList(IComparer, Int32) SortedList(IComparer, Int32)**

Initializes a new instance of the [SortedList](#) class that is empty, has the specified initial capacity, and is sorted according to the specified [IComparer](#) interface.

```
public SortedList (System.Collections.IComparer comparer, int capacity);
new System.Collections.SortedList : System.Collections.IComparer * int ->
System.Collections.SortedList
```

## Parameters

comparer [IComparer](#) [IComparer](#)

The [IComparer](#) implementation to use when comparing keys.

-or-

`null` to use the [Comparable](#) implementation of each key.

capacity [Int32](#) [Int32](#)

The initial number of elements that the [SortedList](#) object can contain.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`capacity` is less than zero.

[OutOfMemoryException](#) [OutOfMemoryException](#)

There is not enough available memory to create a [SortedList](#) object with the specified `capacity`.

## Examples

The following code example creates collections using different [SortedList](#) constructors and demonstrates the differences in the behavior of the collections.

```
using System;
using System.Collections;
using System.Globalization;

public class SamplesSortedList
{

    public static void Main()
    {

        // Create a SortedList using the default comparer.
        SortedList mySL1 = new SortedList( 3 );
        Console.WriteLine("mySL1 (default):");
        mySL1.Add("FIRST", "Hello");
        mySL1.Add("SECOND", "World");
        mySL1.Add("THIRD", "!");
        try
        {
            mySL1.Add("first", "Ola!");
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e);
        }
        PrintKeysAndValues(mySL1);

        // Create a SortedList using the specified case-insensitive comparer.
        SortedList mySL2 = new SortedList(new CaseInsensitiveComparer(), 3);
        Console.WriteLine("mySL2 (case-insensitive comparer):");
        mySL2.Add("FIRST", "Hello");
        mySL2.Add("SECOND", "World");
```

```

mySL2.Add("THIRD", "!");
try
{
    mySL2.Add("first", "Ola!");
}
catch (ArgumentException e)
{
    Console.WriteLine(e);
}
PrintKeysAndValues(mySL2);

// Create a SortedList using the specified CaseInsensitiveComparer,
// which is based on the Turkish culture (tr-TR), where "I" is not
// the uppercase version of "i".
CultureInfo myCul = new CultureInfo("tr-TR");
SortedList mySL3 =
    new SortedList(new CaseInsensitiveComparer(myCul), 3);

Console.WriteLine(
    "mySL3 (case-insensitive comparer, Turkish culture):");

mySL3.Add("FIRST", "Hello");
mySL3.Add("SECOND", "World");
mySL3.Add("THIRD", "!");
try
{
    mySL3.Add("first", "Ola!");
}
catch (ArgumentException e)
{
    Console.WriteLine(e);
}
PrintKeysAndValues(mySL3);

// Create a SortedList using the
// StringComparer.InvariantCultureIgnoreCase value.
SortedList mySL4 = new SortedList(
    StringComparer.InvariantCultureIgnoreCase, 3);

Console.WriteLine("mySL4 (InvariantCultureIgnoreCase):");
mySL4.Add("FIRST", "Hello");
mySL4.Add("SECOND", "World");
mySL4.Add("THIRD", "!");
try
{
    mySL4.Add("first", "Ola!");
}
catch (ArgumentException e)
{
    Console.WriteLine(e);
}
PrintKeysAndValues(mySL4);

}

public static void PrintKeysAndValues(SortedList myList)
{
    Console.WriteLine("      -KEY-      -VALUE-");
    for (int i = 0; i < myList.Count; i++)
    {
        Console.WriteLine("      {0,-6}: {1}",
            myList.GetKey(i), myList.GetByIndex(i));
    }
    Console.WriteLine();
}

```

```

/*
This code produces the following output.
Results vary depending on the system's culture settings.

mySL1 (default):
    -KEY-      -VALUE-
    first : Ola!
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL2 (case-insensitive comparer):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-      -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL3 (case-insensitive comparer, Turkish culture):
    -KEY-      -VALUE-
    FIRST : Hello
    first : Ola!
    SECOND: World
    THIRD : !

mySL4 (InvariantCultureIgnoreCase):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-      -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

*/

```

## Remarks

The elements are sorted according to the specified [IComparer](#) implementation. If the `comparer` parameter is `null`, the [IComparable](#) implementation of each key is used; therefore, each key must implement the [IComparable](#) interface to be capable of comparisons with every other key in the [SortedList](#) object.

The capacity of a [SortedList](#) object is the number of elements that the [SortedList](#) can hold. As elements are added to a [SortedList](#), 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](#) object.

This constructor is an  $O(n)$  operation, where `n` is `capacity`.

See

[IComparer](#)  
[Comparer](#)

Also

[Capacity](#)  
[Capacity](#)

[Performing Culture-Insensitive String Operations in Collections](#)

## **SortedList(IDictionary, IComparer) SortedList(IDictionary,**

# IComparer

Initializes a new instance of the [SortedList](#) class that contains elements copied from the specified dictionary, has the same initial capacity as the number of elements copied, and is sorted according to the specified [IComparer](#) interface.

```
public SortedList (System.Collections.IDictionary d, System.Collections.IComparer comparer);  
new System.Collections.SortedList : System.Collections.IDictionary * System.Collections.IComparer ->  
System.Collections.SortedList
```

Parameters

d [IDictionary](#) [IDictionary](#)

The [IDictionary](#) implementation to copy to a new [SortedList](#) object.

comparer [IComparer](#) [IComparer](#)

The [IComparer](#) implementation to use when comparing keys.

-or-

`null` to use the [IComparable](#) implementation of each key.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`d` is `null`.

[InvalidOperationException](#) [InvalidOperationException](#)

`comparer` is `null`, and one or more elements in `d` do not implement the [IComparable](#) interface.

Examples

The following code example creates collections using different [SortedList](#) constructors and demonstrates the differences in the behavior of the collections.

```
using System;  
using System.Collections;  
using System.Globalization;  
  
public class SamplesSortedList  
{  
  
    public static void Main()  
    {  
  
        // Create the dictionary.  
        Hashtable myHT = new Hashtable();  
        myHT.Add("FIRST", "Hello");  
        myHT.Add("SECOND", "World");  
        myHT.Add("THIRD", "!");  
  
        // Create a SortedList using the default comparer.  
        SortedList mySL1 = new SortedList(myHT);  
        Console.WriteLine("mySL1 (default):");  
        try  
        {  
            mySL1.Add("first", "Ola!");  
        }  
        catch (ArgumentException e)  
        {  
            Console.WriteLine(e.Message);  
        }  
    }  
}
```

```

        Console.WriteLine(e);
    }
    PrintKeysAndValues(mySL1);

    // Create a SortedList using the specified case-insensitive comparer.
    SortedList mySL2 = new SortedList(myHT, new CaseInsensitiveComparer());
    Console.WriteLine("mySL2 (case-insensitive comparer):");
    try
    {
        mySL2.Add("first", "Ola!");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine(e);
    }
    PrintKeysAndValues(mySL2);

    // Create a SortedList using the specified CaseInsensitiveComparer,
    // which is based on the Turkish culture (tr-TR), where "I" is not
    // the uppercase version of "i".
    CultureInfo myCul = new CultureInfo("tr-TR");
    SortedList mySL3 = new SortedList(myHT, new CaseInsensitiveComparer(myCul));
    Console.WriteLine("mySL3 (case-insensitive comparer, Turkish culture):");
    try
    {
        mySL3.Add("first", "Ola!");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine(e);
    }
    PrintKeysAndValues(mySL3);

    // Create a SortedList using the
    // StringComparer.InvariantCultureIgnoreCase value.
    SortedList mySL4 = new SortedList(
        myHT, StringComparer.InvariantCultureIgnoreCase);

    Console.WriteLine("mySL4 (InvariantCultureIgnoreCase):");
    try
    {
        mySL4.Add("first", "Ola!");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine(e);
    }
    PrintKeysAndValues(mySL4);

}

public static void PrintKeysAndValues(SortedList myList)
{
    Console.WriteLine("      -KEY-      -VALUE-");
    for (int i = 0; i < myList.Count; i++)
    {
        Console.WriteLine("      {0,-6}: {1}",
            myList.GetKey(i), myList.GetByIndex(i));
    }
    Console.WriteLine();
}
}

/*
This code produces the following output. Results vary depending on the system's culture settings.

```

```

mySL1 (default):
    -KEY-    -VALUE-
    first : Ola!
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL2 (case-insensitive comparer):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-    -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

mySL3 (case-insensitive comparer, Turkish culture):
    -KEY-    -VALUE-
    FIRST : Hello
    first : Ola!
    SECOND: World
    THIRD : !

mySL4 (InvariantCultureIgnoreCase):
System.ArgumentException: Item has already been added. Key in dictionary: 'FIRST' Key being added:
'first'
    at System.Collections.SortedList.Add(Object key, Object value)
    at SamplesSortedList.Main()
    -KEY-    -VALUE-
    FIRST : Hello
    SECOND: World
    THIRD : !

*/

```

## Remarks

The elements are sorted according to the specified [IComparer](#) implementation. If the `comparer` parameter is `null`, the [IComparable](#) implementation of each key is used; therefore, each key must implement the [IComparable](#) interface to be capable of comparisons with every other key in the [SortedList](#) object.

A [Hashtable](#) object is an example of an [IDictionary](#) implementation that can be passed to this constructor. The new [SortedList](#) object contains a copy of the keys and values stored in the [Hashtable](#).

The capacity of a [SortedList](#) object is the number of elements that the [SortedList](#) can hold. As elements are added to a [SortedList](#), 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](#) object.

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

See

[IDictionary](#)  
[IDictionary](#)

Also

[IComparer](#)  
[IComparer](#)

[Capacity](#)  
[Capacity](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# SortedList.Synchronized

## In this Article

Returns a synchronized (thread-safe) wrapper for a [SortedList](#) object.

```
public static System.Collections.SortedList Synchronized (System.Collections.SortedList list);  
static member Synchronized : System.Collections.SortedList -> System.Collections.SortedList
```

### Parameters

list [SortedList](#) [SortedList](#)

The [SortedList](#) object to synchronize.

### Returns

[SortedList](#) [SortedList](#)

A synchronized (thread-safe) wrapper for the [SortedList](#) object.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`list` is `null`.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) property during the entire enumeration.

```
SortedList myCollection = new SortedList();  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

This method is an O(1) operation.

The following code example shows how to synchronize a [SortedList](#) object, determine whether a [SortedList](#) is synchronized, and use a synchronized [SortedList](#).

```

using System;
using System.Collections;
public class SamplesSortedList {
    public static void Main() {
        // Creates and initializes a new SortedList.
        SortedList mySL = new SortedList();
        mySL.Add( 2, "two" );
        mySL.Add( 3, "three" );
        mySL.Add( 1, "one" );
        mySL.Add( 0, "zero" );
        mySL.Add( 4, "four" );

        // Creates a synchronized wrapper around the SortedList.
        SortedList mySyncdSL = SortedList.Synchronized( mySL );

        // Displays the synchronization status of both SortedLists.
        Console.WriteLine( "mySL is {0}."., mySL.IsSynchronized ? "synchronized" : "not synchronized"
);
        Console.WriteLine( "mySyncdSL is {0}."., mySyncdSL.IsSynchronized ? "synchronized" : "not
synchronized" );
    }
}
/*
This code produces the following output.

mySL is not synchronized.
mySyncdSL is synchronized.
*/

```

## Remarks

To guarantee the thread safety of a [SortedList](#) object, all operations must be done through this wrapper only.

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 causes 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.

See

[IsSynchronized](#)

Also

[SyncRoot](#)

# SortedList.SyncRoot SortedList.SyncRoot

## In this Article

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

```
public virtual object SyncRoot { get; }  
member this.SyncRoot : obj
```

Returns

[Object Object](#)

An object that can be used to synchronize access to the [SortedList](#) object.

## Examples

The following code example shows how to lock the collection using the [SyncRoot](#) property during the entire enumeration.

```
SortedList myCollection = new SortedList();  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

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

## Remarks

To create a synchronized version of the [SortedList](#) object, use the [Synchronized](#) method. However, derived classes can provide their own synchronized version of the [SortedList](#) using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) of the [SortedList](#), not directly on the [SortedList](#). This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [SortedList](#) object.

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 causes 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.

See

[IsSynchronized](#)[IsSynchronized](#)

Also

[Synchronized\(SortedList\)](#)[Synchronized\(SortedList\)](#)

# SortedList.TrimToSize SortedList.TrimToSize

## In this Article

Sets the capacity to the actual number of elements in a [SortedList](#) object.

```
public virtual void TrimToSize ();  
  
abstract member TrimToSize : unit -> unit  
override this.TrimToSize : unit -> unit
```

## Exceptions

[NotSupportedException](#) [NotSupportedException](#)

The [SortedList](#) object is read-only.

-or-

The [SortedList](#) has a fixed size.

## Examples

The following code example shows how to trim the unused portions of a [SortedList](#) object and how to clear its values.

```
using System;  
using System.Collections;  
public class SamplesSortedList {  
  
    public static void Main() {  
  
        // Creates and initializes a new SortedList.  
        SortedList mySL = new SortedList();  
        mySL.Add( "one", "The" );  
        mySL.Add( "two", "quick" );  
        mySL.Add( "three", "brown" );  
        mySL.Add( "four", "fox" );  
        mySL.Add( "five", "jumps" );  
  
        // Displays the count, capacity and values of the SortedList.  
        Console.WriteLine( "Initially," );  
        Console.WriteLine( "    Count      : {0}", mySL.Count );  
        Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );  
        Console.WriteLine( "    Values:" );  
        PrintKeysAndValues( mySL );  
  
        // Trims the SortedList.  
        mySL.TrimToSize();  
  
        // Displays the count, capacity and values of the SortedList.  
        Console.WriteLine( "After TrimToSize," );  
        Console.WriteLine( "    Count      : {0}", mySL.Count );  
        Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );  
        Console.WriteLine( "    Values:" );  
        PrintKeysAndValues( mySL );  
  
        // Clears the SortedList.  
        mySL.Clear();  
  
        // Displays the count, capacity and values of the SortedList.  
        Console.WriteLine( "After Clear," );  
        Console.WriteLine( "    Count      : {0}", mySL.Count );  
        Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );  
        Console.WriteLine( "    Values:" );
```

```

PrintKeysAndValues( mySL );

// Trims the SortedList again.
mySL.TrimToSize();

// Displays the count, capacity and values of the SortedList.
Console.WriteLine( "After the second TrimToSize," );
Console.WriteLine( "    Count      : {0}", mySL.Count );
Console.WriteLine( "    Capacity   : {0}", mySL.Capacity );
Console.WriteLine( "    Values:" );
PrintKeysAndValues( mySL );
}

public static void PrintKeysAndValues( SortedList myList ) {
    Console.WriteLine( "    -KEY-    -VALUE-" );
    for ( int i = 0; i < myList.Count; i++ ) {
        Console.WriteLine( "    {0}:    {1}", myList.GetKey(i), myList.GetByIndex(i) );
    }
    Console.WriteLine();
}
/*
This code produces the following output.

Initially,
Count      : 5
Capacity   : 16
Values:
-KEY-    -VALUE-
five:    jumps
four:    fox
one:    The
three:   brown
two:    quick

After TrimToSize,
Count      : 5
Capacity   : 5
Values:
-KEY-    -VALUE-
five:    jumps
four:    fox
one:    The
three:   brown
two:    quick

After Clear,
Count      : 0
Capacity   : 16
Values:
-KEY-    -VALUE-

After the second TrimToSize,
Count      : 0
Capacity   : 16
Values:
-KEY-    -VALUE-
*/

```

## Remarks

This method can be used to minimize a collection's memory overhead if no new elements will be added to the collection.

To reset a [SortedList](#) object to its initial state, call the [Clear](#) method before calling [TrimToSize](#). Trimming an empty [SortedList](#) sets the capacity of the [SortedList](#) to the default capacity.

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

See

Also

[Clear\(\)](#)  
[Clear\(\)](#)

[Capacity](#)  
[Capacity](#)

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

# SortedList.Values SortedList.Values

## In this Article

Gets the values in a [SortedList](#) object.

```
public virtual System.Collections.ICollection Values { get; }  
member this.Values : System.Collections.ICollection
```

Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) object containing the values in the [SortedList](#) object.

## Remarks

The [ICollection](#) object is a read-only view of the values of the [SortedList](#) object. Modifications made to the underlying [SortedList](#) are immediately reflected in the [ICollection](#).

The elements of the [ICollection](#) are sorted in the same order as the values of the [SortedList](#).

This property is similar to the [GetValueList](#) method, but returns an [ICollection](#) object instead of an [IList](#) object.

This method is an O(1) operation.

See

[ICollection](#)  
[ICollection](#)

Also

[Keys](#)  
[Keys](#)

[GetValueList\(\)](#)  
[GetValueList\(\)](#)

[Performing Culture-Insensitive String Operations in Collections](#)

# Stack Stack Class

Represents a simple last-in-first-out (LIFO) non-generic collection of objects.

## Declaration

```
[System.Runtime.InteropServices.ComVisible(true)]
[Serializable]
public class Stack : ICloneable, System.Collections.ICollection

type Stack = class
    interface ICollection
    interface ICloneable
    interface IEnumerable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

The capacity of a [Stack](#) is the number of elements the [Stack](#) can hold. As elements are added to a [Stack](#), the capacity is automatically increased as required through reallocation.

**Important**

We don't recommend that you use the [Stack](#) class for new development. Instead, we recommend that you use the generic [System.Collections.Generic.Stack<T>](#) class. For more information, see [Non-generic collections shouldn't be used](#) on GitHub.

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](#) accepts [null](#) as a valid value and allows duplicate elements.

## Constructors

[Stack\(\)](#)

[Stack\(\)](#)

Initializes a new instance of the [Stack](#) class that is empty and has the default initial capacity.

[Stack\(ICollection\)](#)

[Stack\(ICollection\)](#)

Initializes a new instance of the [Stack](#) class that contains elements copied from the specified collection and has the same initial capacity as the number of elements copied.

[Stack\(Int32\)](#)

[Stack\(Int32\)](#)

Initializes a new instance of the [Stack](#) class that is empty and has the specified initial capacity or the default initial capacity, whichever is greater.

# Properties

Count

Count

Gets the number of elements contained in the [Stack](#).

IsSynchronized

IsSynchronized

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

SyncRoot

SyncRoot

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

# Methods

Clear()

Clear()

Removes all objects from the [Stack](#).

Clone()

Clone()

Creates a shallow copy of the [Stack](#).

Contains(Object)

Contains(Object)

Determines whether an element is in the [Stack](#).

CopyTo(Array, Int32)

CopyTo(Array, Int32)

Copies the [Stack](#) to an existing one-dimensional [Array](#), starting at the specified array index.

GetEnumerator()

GetEnumerator()

Returns an [IEnumerator](#) for the [Stack](#).

Peek()

`Peek()`

Returns the object at the top of the [Stack](#) without removing it.

`Pop()`

`Pop()`

Removes and returns the object at the top of the [Stack](#).

`Push(Object)`

`Push(Object)`

Inserts an object at the top of the [Stack](#).

`Synchronized(Stack)`

`Synchronized(Stack)`

Returns a synchronized (thread safe) wrapper for the [Stack](#).

`ToArray()`

`ToArray()`

Copies the [Stack](#) to a new array.

## 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.

To guarantee the thread safety of the [Stack](#), all operations must be done through the wrapper returned by the [Synchronized\(Stack\)](#) method.

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 causes 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.

# Stack.Clear Stack.Clear

## In this Article

Removes all objects from the [Stack](#).

```
public virtual void Clear ();  
  
abstract member Clear : unit -> unit  
override this.Clear : unit -> unit
```

## Examples

The following example shows how to clear the values of the [Stack](#).

```

using System;
using System.Collections;

public class SamplesStack {
    public static void Main() {
        // Creates and initializes a new Stack.
        Stack myStack = new Stack();
        myStack.Push( "The" );
        myStack.Push( "quick" );
        myStack.Push( "brown" );
        myStack.Push( "fox" );
        myStack.Push( "jumps" );

        // Displays the count and values of the Stack.
        Console.WriteLine( "Initially," );
        Console.WriteLine( "    Count      : {0}", myStack.Count );
        Console.Write( "    Values:" );
        PrintValues( myStack );

        // Clears the Stack.
        myStack.Clear();

        // Displays the count and values of the Stack.
        Console.WriteLine( "After Clear," );
        Console.WriteLine( "    Count      : {0}", myStack.Count );
        Console.Write( "    Values:" );
        PrintValues( myStack );
    }

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

/*
This code produces the following output.

Initially,
    Count      : 5
    Values:    jumps    fox    brown    quick    The
After Clear,
    Count      : 0
    Values:
*/

```

## Remarks

**Count** is set to zero, and references to other objects from elements of the collection are also released.

This method is an O( $n$ ) operation, where  $n$  is **Count**.

# Stack.Clone Stack.Clone

## In this Article

Creates a shallow copy of the [Stack](#).

```
public virtual object Clone ();  
  
abstract member Clone : unit -> obj  
override this.Clone : unit -> obj
```

Returns

[Object Object](#)

A shallow copy of the [Stack](#).

## Remarks

A shallow copy of a collection copies only the elements of the collection, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new collection point to the same objects that the references in the original collection point to.

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](#).

# Stack.Contains Stack.Contains

## In this Article

Determines whether an element is in the [Stack](#).

```
public virtual bool Contains (object obj);  
  
abstract member Contains : obj -> bool  
override this.Contains : obj -> bool
```

## Parameters

obj [Object](#) [Object](#)

The object to locate in the [Stack](#). The value can be [null](#).

## Returns

[Boolean](#) [Boolean](#)

[true](#), if [obj](#) is found in the [Stack](#); otherwise, [false](#).

## Remarks

This method determines equality by calling the [Object.Equals](#) method.

This method performs a linear search; therefore, this method is an  $O(n)$  operation, where  $n$  is [Count](#).

Starting with the .NET Framework 2.0, this method tests for equality by passing the [obj](#) argument to the [Equals](#) method of individual objects in the collection. In the earlier versions of the .NET Framework, this determination was made by using passing the individual items in the collection to the [Equals](#) method of the [obj](#) argument.

See

[Performing Culture-Insensitive String Operations](#)

Also

# Stack.CopyTo Stack.CopyTo

## In this Article

Copies the [Stack](#) to an existing one-dimensional [Array](#), starting at the specified array index.

```
public virtual void CopyTo (Array array, int index);  
  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

## Parameters

array [Array](#) [Array](#)

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

index [Int32](#) [Int32](#)

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

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

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

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

[index](#) is less than zero.

[ArgumentException](#) [ArgumentException](#)

[array](#) is multidimensional.

-or-

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

[InvalidCastException](#) [InvalidCastException](#)

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

## Examples

The following example shows how to copy a [Stack](#) into a one-dimensional array.

```
using System;  
using System.Collections;  
public class SamplesStack {  
  
    public static void Main() {  
  
        // Creates and initializes the source Stack.  
        Stack mySourceQ = new Stack();  
        mySourceQ.Push( "barn" );  
        mySourceQ.Push( "the" );  
        mySourceQ.Push( "in" );  
        mySourceQ.Push( "cats" );  
        mySourceQ.Push( "napping" );  
        mySourceQ.Push( "three" );
```

```

// Creates and initializes the one-dimensional target Array.
Array myTargetArray=Array.CreateInstance( typeof(String), 15 );
myTargetArray.SetValue( "The", 0 );
myTargetArray.SetValue( "quick", 1 );
myTargetArray.SetValue( "brown", 2 );
myTargetArray.SetValue( "fox", 3 );
myTargetArray.SetValue( "jumps", 4 );
myTargetArray.SetValue( "over", 5 );
myTargetArray.SetValue( "the", 6 );
myTargetArray.SetValue( "lazy", 7 );
myTargetArray.SetValue( "dog", 8 );

// Displays the values of the target Array.
Console.WriteLine( "The target Array contains the following (before and after copying):" );
PrintValues( myTargetArray, ' ' );

// Copies the entire source Stack to the target Array, starting at index 6.
mySourceQ.CopyTo( myTargetArray, 6 );

// Displays the values of the target Array.
PrintValues( myTargetArray, ' ' );

// Copies the entire source Stack to a new standard array.
Object[] myStandardArray = mySourceQ.ToArray();

// Displays the values of the new standard array.
Console.WriteLine( "The new standard array contains the following:" );
PrintValues( myStandardArray, ' ' );

}

public static void PrintValues( Array myArr, char mySeparator ) {
    foreach ( Object myObj in myArr ) {
        Console.Write( "{0}{1}", mySeparator, myObj );
    }
    Console.WriteLine();
}

/*
This code produces the following output.

The target Array contains the following (before and after copying):
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over three napping cats in the barn
The new standard array contains the following:
three napping cats in the barn
*/

```

## 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](#).

See

[ToArray\(\)](#)

Also

# Stack.Count Stack.Count

## In this Article

Gets the number of elements contained in the [Stack](#).

```
public virtual int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements contained in the [Stack](#).

## Remarks

The capacity is the number of elements that the [Stack](#) can store. [Count](#) is the number of elements that are actually in the [Stack](#).

The capacity is always greater than or equal to [Count](#). If [Count](#) exceeds the capacity while adding elements, the capacity is automatically increased by 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.

# Stack.GetEnumerator Stack.GetEnumerator

## In this Article

Returns an [IEnumerator](#) for the [Stack](#).

```
public virtual System.Collections.IEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the [Stack](#).

## Remarks

[Visual Basic, C#]

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 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.

This method is an O(1) operation.

See

[IEnumerator](#)[IEnumerator](#)

Also

# Stack.IsSynchronized Stack.IsSynchronized

## In this Article

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

```
public virtual bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean](#)

`true`, if access to the [Stack](#) is synchronized (thread safe); otherwise, `false`. The default is `false`.

## Examples

The following example shows how to synchronize a [Stack](#), determine if a [Stack](#) is synchronized, and use a synchronized [Stack](#).

```
using System;  
using System.Collections;  
public class SamplesStack {  
  
    public static void Main() {  
  
        // Creates and initializes a new Stack.  
        Stack myStack = new Stack();  
        myStack.Push( "The" );  
        myStack.Push( "quick" );  
        myStack.Push( "brown" );  
        myStack.Push( "fox" );  
  
        // Creates a synchronized wrapper around the Stack.  
        Stack mySyncdStack = Stack.Synchronized( myStack );  
  
        // Displays the synchronization status of both stacks.  
        Console.WriteLine( "myStack is {0}." ,  
            myStack.IsSynchronized ? "synchronized" : "not synchronized" );  
        Console.WriteLine( "mySyncdStack is {0}." ,  
            mySyncdStack.IsSynchronized ? "synchronized" : "not synchronized" );  
    }  
}  
/*  
This code produces the following output.  
  
myStack is not synchronized.  
mySyncdStack is synchronized.  
*/
```

## Remarks

To guarantee the thread safety of the [Stack](#), all operations must be done through the wrapper returned by the [Synchronized](#) method.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
Stack myCollection = new Stack();

lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

# Stack.Peek Stack.Peek

## In this Article

Returns the object at the top of the [Stack](#) without removing it.

```
public virtual object Peek ();  
  
abstract member Peek : unit -> obj  
override this.Peek : unit -> obj
```

Returns

[Object Object](#)

The [Object](#) at the top of the [Stack](#).

Exceptions

[InvalidOperationException InvalidOperationException](#)

The [Stack](#) is empty.

## Examples

The following example shows how to add elements to the [Stack](#), remove elements from the [Stack](#), or view the element at the top of the [Stack](#).

```

using System;
using System.Collections;
public class SamplesStack {

    public static void Main() {

        // Creates and initializes a new Stack.
        Stack myStack = new Stack();
        myStack.Push( "The" );
        myStack.Push( "quick" );
        myStack.Push( "brown" );
        myStack.Push( "fox" );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Removes an element from the Stack.
        Console.WriteLine( "(Pop) {0}", myStack.Pop() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Removes another element from the Stack.
        Console.WriteLine( "(Pop) {0}", myStack.Pop() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Views the first element in the Stack but does not remove it.
        Console.WriteLine( "(Peek) {0}", myStack.Peek() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );
    }

    public static void PrintValues( IEnumerable myCollection, char mySeparator ) {
        foreach ( Object obj in myCollection )
            Console.Write( "{0}{1}", mySeparator, obj );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Stack values: fox brown quick The
(Pop) fox
Stack values: brown quick The
(Pop) brown
Stack values: quick The
(Peek) quick
Stack values: quick The
*/

```

## Remarks

This method is similar to the [Pop](#) method, but [Peek](#) does not modify the [Stack](#).

`null` can be pushed onto the [Stack](#) as a placeholder, if needed. To distinguish between a null value and the end of the stack, check the [Count](#) property or catch the [InvalidOperationException](#), which is thrown when the [Stack](#) is empty.

This method is an O(1) operation.

See

Also

[Pop\(\)Pop\(\)](#)

[Push\(Object\)Push\(Object\)](#)

# Stack.Pop Stack.Pop

## In this Article

Removes and returns the object at the top of the [Stack](#).

```
public virtual object Pop ();  
  
abstract member Pop : unit -> obj  
override this.Pop : unit -> obj
```

Returns

[Object Object](#)

The [Object](#) removed from the top of the [Stack](#).

Exceptions

[InvalidOperationException InvalidOperationException](#)

The [Stack](#) is empty.

## Examples

The following example shows how to add elements to the [Stack](#), remove elements from the [Stack](#), or view the element at the top of the [Stack](#).

```

using System;
using System.Collections;
public class SamplesStack {

    public static void Main() {

        // Creates and initializes a new Stack.
        Stack myStack = new Stack();
        myStack.Push( "The" );
        myStack.Push( "quick" );
        myStack.Push( "brown" );
        myStack.Push( "fox" );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Removes an element from the Stack.
        Console.WriteLine( "(Pop) {0}", myStack.Pop() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Removes another element from the Stack.
        Console.WriteLine( "(Pop) {0}", myStack.Pop() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Views the first element in the Stack but does not remove it.
        Console.WriteLine( "(Peek) {0}", myStack.Peek() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );
    }

    public static void PrintValues( IEnumerable myCollection, char mySeparator ) {
        foreach ( Object obj in myCollection )
            Console.Write( "{0}{1}", mySeparator, obj );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Stack values: fox brown quick The
(Pop) fox
Stack values: brown quick The
(Pop) brown
Stack values: quick The
(Peek) quick
Stack values: quick The
*/

```

## Remarks

This method is similar to the [Peek](#) method, but [Peek](#) does not modify the [Stack](#).

`null` can be pushed onto the [Stack](#) as a placeholder, if needed. To distinguish between a null value and the end of the stack, check the [Count](#) property or catch the [InvalidOperationException](#), which is thrown when the [Stack](#) is empty.

This method is an O(1) operation.

See

Also

[Peek\(\)](#)[Peek\(\)](#)

[Push\(Object\)](#)[Push\(Object\)](#)

# Stack.Push Stack.Push

## In this Article

Inserts an object at the top of the [Stack](#).

```
public virtual void Push (object obj);  
  
abstract member Push : obj -> unit  
override this.Push : obj -> unit
```

## Parameters

obj	<a href="#">Object</a>
-----	------------------------

The [Object](#) to push onto the [Stack](#). The value can be `null`.

## Examples

The following example shows how to add elements to the [Stack](#), remove elements from the [Stack](#), or view the element at the top of the [Stack](#).

```

using System;
using System.Collections;
public class SamplesStack {

    public static void Main() {

        // Creates and initializes a new Stack.
        Stack myStack = new Stack();
        myStack.Push( "The" );
        myStack.Push( "quick" );
        myStack.Push( "brown" );
        myStack.Push( "fox" );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Removes an element from the Stack.
        Console.WriteLine( "(Pop) {0}", myStack.Pop() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Removes another element from the Stack.
        Console.WriteLine( "(Pop) {0}", myStack.Pop() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );

        // Views the first element in the Stack but does not remove it.
        Console.WriteLine( "(Peek) {0}", myStack.Peek() );

        // Displays the Stack.
        Console.Write( "Stack values:" );
        PrintValues( myStack, ' ' );
    }

    public static void PrintValues( IEnumerable myCollection, char mySeparator ) {
        foreach ( Object obj in myCollection )
            Console.Write( "{0}{1}", mySeparator, obj );
        Console.WriteLine();
    }
}

/*
This code produces the following output.

Stack values: fox brown quick The
(Pop) fox
Stack values: brown quick The
(Pop) brown
Stack values: quick The
(Peek) quick
Stack values: quick The
*/

```

## Remarks

If `Count` already equals the capacity, the capacity of the `Stack` is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

`null` can be pushed onto the `Stack` 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`.

See

[Peek\(\)Peek\(\)](#)

Also

[Pop\(\)Pop\(\)](#)

# Stack Stack

In this Article

## Overloads

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

### Stack()

Initializes a new instance of the [Stack](#) class that is empty and has the default initial capacity.

```
public Stack ();
```

#### Remarks

The capacity of a [Stack](#) is the number of elements that the [Stack](#) can hold. As elements are added to a [Stack](#), 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](#).

This constructor is an O(1) operation.

### Stack(ICollection) Stack(ICollection)

Initializes a new instance of the [Stack](#) class that contains elements copied from the specified collection and has the same initial capacity as the number of elements copied.

```
public Stack (System.Collections.ICollection col);  
new System.Collections.Stack : System.Collections.ICollection -> System.Collections.Stack
```

#### Parameters

col

[ICollection](#) [ICollection](#)

The [ICollection](#) to copy elements from.

#### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`col` is `null`.

#### Remarks

The capacity of a [Stack](#) is the number of elements that the [Stack](#) can hold. As elements are added to a [Stack](#), 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](#).

The elements are copied onto the [Stack](#) in the same order they are read by the [IEnumerator](#) of the [ICollection](#).

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

See

[ICollection](#)  
[Collection](#)

Also

[IEnumerator](#)  
[Enumerator](#)

## Stack(Int32) Stack(Int32)

Initializes a new instance of the [Stack](#) class that is empty and has the specified initial capacity or the default initial capacity, whichever is greater.

```
public Stack (int initialCapacity);  
new System.Collections.Stack : int -> System.Collections.Stack
```

Parameters

initialCapacity [Int32](#) [Int32](#)

The initial number of elements that the [Stack](#) can contain.

Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

`initialCapacity` is less than zero.

Remarks

The capacity of a [Stack](#) is the number of elements that the [Stack](#) can hold. As elements are added to a [Stack](#), 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](#).

This constructor is an  $O(n)$  operation, where  $n$  is `initialCapacity`.

# Stack.Synchronized Stack.Synchronized

## In this Article

Returns a synchronized (thread safe) wrapper for the [Stack](#).

```
public static System.Collections.Stack Synchronized (System.Collections.Stack stack);  
static member Synchronized : System.Collections.Stack -> System.Collections.Stack
```

## Parameters

stack [Stack](#) [Stack](#)

The [Stack](#) to synchronize.

## Returns

[Stack](#) [Stack](#)

A synchronized wrapper around the [Stack](#).

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`stack` is `null`.

## Examples

The following example shows how to synchronize a [Stack](#), determine if a [Stack](#) is synchronized, and use a synchronized [Stack](#).

```
using System;  
using System.Collections;  
public class SamplesStack {  
  
    public static void Main() {  
  
        // Creates and initializes a new Stack.  
        Stack myStack = new Stack();  
        myStack.Push( "The" );  
        myStack.Push( "quick" );  
        myStack.Push( "brown" );  
        myStack.Push( "fox" );  
  
        // Creates a synchronized wrapper around the Stack.  
        Stack mySyncdStack = Stack.Synchronized( myStack );  
  
        // Displays the synchronization status of both stacks.  
        Console.WriteLine( "myStack is {0}.",  
            myStack.IsSynchronized ? "synchronized" : "not synchronized" );  
        Console.WriteLine( "mySyncdStack is {0}.",  
            mySyncdStack.IsSynchronized ? "synchronized" : "not synchronized" );  
    }  
}  
/*  
This code produces the following output.  
  
myStack is not synchronized.  
mySyncdStack is synchronized.  
*/
```

## Remarks

To guarantee the thread safety of the [Stack](#), all operations must be done through this wrapper.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
Stack myCollection = new Stack();

lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

This method is an O(1) operation.

# Stack.SyncRoot Stack.SyncRoot

## In this Article

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

```
public virtual object SyncRoot { get; }  
member this.SyncRoot : obj
```

Returns

[Object Object](#)

An [Object](#) that can be used to synchronize access to the [Stack](#).

## Remarks

To create a synchronized version of the [Stack](#), use the [Synchronized](#) method. However, derived classes can provide their own synchronized version of the [Stack](#) using the [SyncRoot](#) property. The synchronizing code must perform operations on the [SyncRoot](#) of the [Stack](#), not directly on the [Stack](#). This ensures proper operation of collections that are derived from other objects. Specifically, it maintains proper synchronization with other threads that might be simultaneously modifying the [Stack](#) object.

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 causes 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 following code example shows how to lock the collection using the [SyncRoot](#) during the entire enumeration.

```
Stack myCollection = new Stack();  
  
lock(myCollection.SyncRoot)  
{  
    foreach (object item in myCollection)  
    {  
        // Insert your code here.  
    }  
}
```

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

# Stack.ToArray Stack.ToArray

## In this Article

Copies the [Stack](#) to a new array.

```
public virtual object[] ToArray ();  
  
abstract member ToArray : unit -> obj[]  
override this.ToArray : unit -> obj[]
```

Returns

[Object\[\]](#)

A new array containing copies of the elements of the [Stack](#).

## Examples

The following example shows how to copy a [Stack](#) into a one-dimensional array.

```
using System;  
using System.Collections;  
public class SamplesStack {  
  
    public static void Main() {  
  
        // Creates and initializes the source Stack.  
        Stack mySourceQ = new Stack();  
        mySourceQ.Push( "barn" );  
        mySourceQ.Push( "the" );  
        mySourceQ.Push( "in" );  
        mySourceQ.Push( "cats" );  
        mySourceQ.Push( "napping" );  
        mySourceQ.Push( "three" );  
  
        // Creates and initializes the one-dimensional target Array.  
        Array myTargetArray=Array.CreateInstance( typeof(String), 15 );  
        myTargetArray.SetValue( "The", 0 );  
        myTargetArray.SetValue( "quick", 1 );  
        myTargetArray.SetValue( "brown", 2 );  
        myTargetArray.SetValue( "fox", 3 );  
        myTargetArray.SetValue( "jumps", 4 );  
        myTargetArray.SetValue( "over", 5 );  
        myTargetArray.SetValue( "the", 6 );  
        myTargetArray.SetValue( "lazy", 7 );  
        myTargetArray.SetValue( "dog", 8 );  
  
        // Displays the values of the target Array.  
        Console.WriteLine( "The target Array contains the following (before and after copying):" );  
        PrintValues( myTargetArray, ' ' );  
  
        // Copies the entire source Stack to the target Array, starting at index 6.  
        mySourceQ.CopyTo( myTargetArray, 6 );  
  
        // Displays the values of the target Array.  
        PrintValues( myTargetArray, ' ' );  
  
        // Copies the entire source Stack to a new standard array.  
        Object[] myStandardArray = mySourceQ.ToArray();  
  
        // Displays the values of the new standard array.  
        Console.WriteLine( "The new standard array contains the following:" );  
        PrintValues( myStandardArray, ' ' );
```

```
}

public static void PrintValues( Array myArr, char mySeparator ) {
    foreach ( Object myObj in myArr ) {
        Console.Write( "{0}{1}", mySeparator, myObj );
    }
    Console.WriteLine();
}

/*
This code produces the following output.

The target Array contains the following (before and after copying):
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over three napping cats in the barn
The new standard array contains the following:
three napping cats in the barn
*/
```

## 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](#).

See

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

Also

[Pop\(\)](#)[Pop\(\)](#)

# StructuralComparisons StructuralComparisons Class

Provides objects for performing a structural comparison of two collection objects.

## Declaration

```
public static class StructuralComparisons  
type StructuralComparisons = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

The [StructuralComparisons](#) class returns the following two predefined comparison objects:

- An [IComparer](#) implementation that can be passed to a method such as [Array.IStructuralComparable.CompareTo\(Object, IComparer\)](#) or [Tuple<T1,T2,T3>.IStructuralComparable.CompareTo\(Object, IComparer\)](#) to perform a structural comparison of two objects. It is designed to indicate whether the first object precedes, follows, or occurs in the same position as the second object in the sort order.
- An [IEqualityComparer](#) implementation that can be passed to a method such as [Array.IStructuralEquatable.Equals\(Object, IEqualityComparer\)](#) or [Tuple<T1,T2,T3>.IStructuralEquatable.Equals\(Object, IEqualityComparer\)](#) to perform a comparison for structural equality.

The objects can be used to perform a structural comparison or a structural equality comparison of two collection objects, such as array or tuple objects. In structural comparison, two objects are compared based on their values. Objects can be ordered based on some criteria, and two objects are considered equal when they have equal values, not because they reference the same physical object

## Properties

[StructuralComparer](#)

[StructuralComparer](#)

Gets a predefined object that performs a structural comparison of two objects.

[StructuralEqualityComparer](#)

[StructuralEqualityComparer](#)

Gets a predefined object that compares two objects for structural equality.

# StructuralComparisons.StructuralComparer StructuralComparisons.StructuralComparer

## In this Article

Gets a predefined object that performs a structural comparison of two objects.

```
public static System.Collections.IComparer StructuralComparer { get; }  
member this.StructuralComparer : System.Collections.IComparer
```

Returns

[IComparer](#)

A predefined object that is used to perform a structural comparison of two collection objects.

## Remarks

When the [IComparer](#) object returned by this property is passed to the comparison method of a collection object, such as [Array.IStructuralComparable.CompareTo\(Object, IComparer\)](#) or [Tuple<T1,T2,T3>.IStructuralComparable.CompareTo\(Object, IComparer\)](#), its [Compare](#) method is called for each member of an array or for each component of a tuple. This implementation of the [Compare](#) method behaves as follows when it compares each item of a collection object with the corresponding item of another collection object:

- It considers two items that are `null` to be equal, and considers a null item to be less than an item that is not null.
- If the first item in the comparison can be cast to an [IStructuralComparable](#) object (in other words, if it is a collection object that implements the [IStructuralComparable](#) interface), it calls the [CompareTo](#) method.
- If the first item in the comparison cannot be cast to an [IStructuralComparable](#) object (in other words, if it is not a collection object that implements the [IStructuralComparable](#) interface), it calls the `Comparer.Default.Compare` method.

# StructuralComparisons.StructuralEqualityComparer

## StructuralComparisons.StructuralEqualityComparer

### In this Article

Gets a predefined object that compares two objects for structural equality.

```
public static System.Collections.IEqualityComparer StructuralEqualityComparer { get; }
```

```
member this.StructuralEqualityComparer : System.Collections.IEqualityComparer
```

Returns

[IEqualityComparer](#)

A predefined object that is used to compare two collection objects for structural equality.

## Remarks

When the [IComparer](#) object returned by this property is passed to the equality comparison method of a collection object, such as [Array.IStructuralEquatable.Equals\(Object, IEqualityComparer\)](#) or [Tuple<T1,T2,T3>.IStructuralEquatable.Equals\(Object, IEqualityComparer\)](#), its [IEqualityComparer.Equals](#) method is called for each member of an array or for each component of a tuple. This implementation of the [Equals](#) method behaves as follows when it compares each item of a collection object with the corresponding item of another collection object:

- If both items are `null`, it considers the two items to be equal.
- If one item is null but the other item is not, it considers the two items to be unequal.
- If the first item in the comparison can be cast to an [IStructuralEquatable](#) object (in other words, if it is a collection object that implements the [IStructuralEquatable](#) interface), it calls the [IStructuralEquatable.Equals](#) method.
- If the first item in the comparison cannot be cast to an [IStructuralEquatable](#) object (in other words, if it is not a collection object that implements the [IStructuralEquatable](#) interface), it calls the item's `Equals` method.