Contents

MoveNext

Reset

System.Collections.Specialized Namespace

The System.Collections.Specialized namespace contains specialized and strongly-typed collections; for example, a linked list dictionary, a bit vector, and collections that contain only strings.

Introduction

Specialized collections are collections with highly specific purposes. NameValueCollection is based on NameObjectCollectionBase; however, NameValueCollection accepts multiple values per key, whereas NameObjectCollectionBase accepts only one value per key.

Some strongly typed collections in the System.Collections.Specialized namespace are StringCollection and StringDictionary, both of which contain values that are exclusively strings.

The CollectionsUtil class creates instances of case-insensitive collections.

Some collections transform. For example, the HybridDictionary class starts as a ListDictionary and becomes a Hashtable when it becomes large. The KeyedCollection<TKey,TItem> is a list but it also creates a lookup dictionary when the number of elements reaches a specified threshold.

Classes

`CollectionChangedEventManager`	Provides a WeakEventManager implementation so that you can use the "weak event listener" pattern to attach listeners for the CollectionChanged event.
`CollectionsUtil`	Creates collections that ignore the case in strings.
`HybridDictionary`	Implements `IDictionary` by using a ListDictionary while the collection is small, and then switching to a Hashtable when the collection gets large.
`ListDictionary`	Implements `IDictionary` using a singly linked list. Recommended for collections that typically include fewer than 10 items.
`NameObjectCollectionBase`	Provides the `abstract` base class for a collection of associated String keys and Object values that can be accessed either with the key or with the index.
`NameObjectCollectionBase.KeysCollection`	Represents a collection of the String keys of a collection.
`NameValueCollection`	Represents a collection of associated String keys and String values that can be accessed either with the key or with the index.

`NotifyCollectionChangedEventArgs`	Provides data for the CollectionChanged event.
`OrderedDictionary`	Represents a collection of key/value pairs that are accessible by the key or index.
`StringCollection`	Represents a collection of strings.
`StringDictionary`	Implements a hash table with the key and the value strongly typed to be strings rather than objects.
`StringEnumerator`	Supports a simple iteration over a StringCollection.

Structs

`BitVector32`	Provides a simple structure that stores Boolean values and small integers in 32 bits of memory.
`BitVector32.Section`	Represents a section of the vector that can contain an integer number.

Interfaces

`INotifyCollectionChanged`	Notifies listeners of dynamic changes, such as when an item is added and removed or the whole list is cleared.
`IOrderedDictionary`	Represents an indexed collection of key/value pairs.

Enums

`NotifyCollectionChangedAction`	Describes the action that caused a CollectionChanged event.

Delegates

`NotifyCollectionChangedEventHandler`	Represents the method that handles the CollectionChanged event.

BitVector32 BitVector32 Struct

Provides a simple structure that stores Boolean values and small integers in 32 bits of memory.

Declaration

```
public struct BitVector32
```

```
type BitVector32 = struct
```

Inheritance Hierarchy

`Object` `Object`

`ValueType` `ValueType`

Remarks

BitVector32 is more efficient than BitArray for Boolean values and small integers that are used internally. A BitArray can grow indefinitely as needed, but it has the memory and performance overhead that a class instance requires. In contrast, a BitVector32 uses only 32 bits.

A BitVector32 structure can be set up to contain either sections for small integers or bit flags for Booleans, but not both. A BitVector32.Section is a window into the BitVector32 and is composed of the smallest number of consecutive bits that can contain the maximum value specified in CreateSection. For example, a section with a maximum value of 1 is composed of only one bit, whereas a section with a maximum value of 5 is composed of three bits. You can create a BitVector32.Section with a maximum value of 1 to serve as a Boolean, thereby allowing you to store integers and Booleans in the same BitVector32.

Some members can be used for a BitVector32 that is set up as sections, while other members can be used for one that is set up as bit flags. For example, the BitVector32.Item[Int32] property is the indexer for a BitVector32 that is set up as sections, and the BitVector32.Item[Int32] property is the indexer for a BitVector32 that is set up as bit flags. CreateMask creates a series of masks that can be used to access individual bits in a BitVector32 that is set up as bit flags.

Using a mask on a BitVector32 that is set up as sections might cause unexpected results.

Constructors

`BitVector32(BitVector32)`

`BitVector32(BitVector32)`

Initializes a new instance of the BitVector32 structure containing the data represented in an existing BitVector32 structure.

`BitVector32(Int32)`

`BitVector32(Int32)`

Initializes a new instance of the BitVector32 structure containing the data represented in an integer.

Properties

`Data`

Data

Gets the value of the BitVector32 as an integer.

Item[BitVector32+Section]

Item[BitVector32+Section]

Gets or sets the value stored in the specified BitVector32.Section.

Item[Int32]

Item[Int32]

Gets or sets the state of the bit flag indicated by the specified mask.

Methods

CreateMask()

CreateMask()

Creates the first mask in a series of masks that can be used to retrieve individual bits in a BitVector32 that is set up as bit flags.

CreateMask(Int32)

CreateMask(Int32)

Creates an additional mask following the specified mask in a series of masks that can be used to retrieve individual bits in a BitVector32 that is set up as bit flags.

CreateSection(Int16)

CreateSection(Int16)

Creates the first BitVector32.Section in a series of sections that contain small integers.

CreateSection(Int16, BitVector32+Section)

CreateSection(Int16, BitVector32+Section)

Creates a new BitVector32.Section following the specified BitVector32.Section in a series of sections that contain small integers.

Equals(Object)

Equals(Object)

Determines whether the specified object is equal to the BitVector32.

GetHashCode()

`GetHashCode()`

Serves as a hash function for the BitVector32.

`ToString()`

`ToString()`

Returns a string that represents the current BitVector32.

`ToString(BitVector32)`

`ToString(BitVector32)`

Returns a string that represents the specified BitVector32.

BitVector32 BitVector32

In this Article

Overloads

BitVector32(BitVector32) BitVector32(BitVector32)	Initializes a new instance of the BitVector32 structure containing the data represented in an existing BitVector32 structure.
BitVector32(Int32) BitVector32(Int32)	Initializes a new instance of the BitVector32 structure containing the data represented in an integer.

BitVector32(BitVector32) BitVector32(BitVector32)

Initializes a new instance of the BitVector32 structure containing the data represented in an existing BitVector32 structure.

```
public BitVector32 (System.Collections.Specialized.BitVector32 value);
```

```
new System.Collections.Specialized.BitVector32 : System.Collections.Specialized.BitVector32 ->
System.Collections.Specialized.BitVector32
```

Parameters

value BitVector32 BitVector32

A BitVector32 structure that contains the data to copy.

Remarks

This constructor is an O(1) operation.

BitVector32(Int32) BitVector32(Int32)

Initializes a new instance of the BitVector32 structure containing the data represented in an integer.

```
public BitVector32 (int data);
```

```
new System.Collections.Specialized.BitVector32 : int -> System.Collections.Specialized.BitVector32
```

Parameters

data Int32 Int32

An integer representing the data of the new BitVector32.

Remarks

This constructor is an O(1) operation.

BitVector32.CreateMask BitVector32.CreateMask

In this Article

Overloads

CreateMask() CreateMask()	Creates the first mask in a series of masks that can be used to retrieve individual bits in a BitVector32 that is set up as bit flags.
CreateMask(Int32) CreateMask(Int32)	Creates an additional mask following the specified mask in a series of masks that can be used to retrieve individual bits in a BitVector32 that is set up as bit flags.

CreateMask() CreateMask()

Creates the first mask in a series of masks that can be used to retrieve individual bits in a BitVector32 that is set up as bit flags.

```
public static int CreateMask ();
```

```
static member CreateMask : unit -> int
```

Returns

Int32 Int32

A mask that isolates the first bit flag in the BitVector32.

Examples

The following code example shows how to create and use masks.

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


public class SamplesBitVector32  {

   public static void Main()  {

      // Creates and initializes a BitVector32 with all bit flags set to FALSE.
      BitVector32 myBV = new BitVector32( 0 );

      // Creates masks to isolate each of the first five bit flags.
      int myBit1 = BitVector32.CreateMask();
      int myBit2 = BitVector32.CreateMask( myBit1 );
      int myBit3 = BitVector32.CreateMask( myBit2 );
      int myBit4 = BitVector32.CreateMask( myBit3 );
      int myBit5 = BitVector32.CreateMask( myBit4 );
      Console.WriteLine( "Initial:                   {0}", myBV.ToString() );

      // Sets the third bit to TRUE.
      myBV[myBit3] = true;
      Console.WriteLine( "myBit3 = TRUE              {0}", myBV.ToString() );

      // Combines two masks to access multiple bits at a time.
      myBV[myBit4 + myBit5] = true;
      Console.WriteLine( "myBit4 + myBit5 = TRUE     {0}", myBV.ToString() );
      myBV[myBit1 | myBit2] = true;
      Console.WriteLine( "myBit1 | myBit2 = TRUE     {0}", myBV.ToString() );

   }

}

/*
This code produces the following output.

Initial:                 BitVector32{00000000000000000000000000000000}
myBit3 = TRUE            BitVector32{00000000000000000000000000000100}
myBit4 + myBit5 = TRUE   BitVector32{00000000000000000000000000011100}
myBit1 | myBit2 = TRUE   BitVector32{00000000000000000000000000011111}

*/
```

Remarks

Use `CreateMask()` to create the first mask in a series and `CreateMask(int)` for all subsequent masks.

Multiple masks can be created to refer to the same bit flag.

The resulting mask isolates only one bit flag in the BitVector32. You can combine masks using the bitwise OR operation to create a mask that isolates multiple bit flags in the BitVector32.

Using a mask on a BitVector32 that is set up as sections might cause unexpected results.

This method is an O(1) operation.

CreateMask(Int32) CreateMask(Int32)

Creates an additional mask following the specified mask in a series of masks that can be used to retrieve individual bits in a BitVector32 that is set up as bit flags.

```
public static int CreateMask (int previous);
```

```
static member CreateMask : int -> int
```

Parameters

previous Int32 Int32

The mask that indicates the previous bit flag.

Returns

Int32 Int32

A mask that isolates the bit flag following the one that `previous` points to in BitVector32.

Exceptions

InvalidOperationException InvalidOperationException

`previous` indicates the last bit flag in the BitVector32.

Examples

The following code example shows how to create and use masks.

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


public class SamplesBitVector32  {

   public static void Main()  {

      // Creates and initializes a BitVector32 with all bit flags set to FALSE.
      BitVector32 myBV = new BitVector32( 0 );

      // Creates masks to isolate each of the first five bit flags.
      int myBit1 = BitVector32.CreateMask();
      int myBit2 = BitVector32.CreateMask( myBit1 );
      int myBit3 = BitVector32.CreateMask( myBit2 );
      int myBit4 = BitVector32.CreateMask( myBit3 );
      int myBit5 = BitVector32.CreateMask( myBit4 );
      Console.WriteLine( "Initial:                   {0}", myBV.ToString() );

      // Sets the third bit to TRUE.
      myBV[myBit3] = true;
      Console.WriteLine( "myBit3 = TRUE              {0}", myBV.ToString() );

      // Combines two masks to access multiple bits at a time.
      myBV[myBit4 + myBit5] = true;
      Console.WriteLine( "myBit4 + myBit5 = TRUE     {0}", myBV.ToString() );
      myBV[myBit1 | myBit2] = true;
      Console.WriteLine( "myBit1 | myBit2 = TRUE     {0}", myBV.ToString() );

   }

}

/*
This code produces the following output.

Initial:                  BitVector32{00000000000000000000000000000000}
myBit3 = TRUE             BitVector32{00000000000000000000000000000100}
myBit4 + myBit5 = TRUE    BitVector32{00000000000000000000000000011100}
myBit1 | myBit2 = TRUE    BitVector32{00000000000000000000000000011111}

*/
```

Remarks

Use `CreateMask()` to create the first mask in a series and `CreateMask(int)` for all subsequent masks.

Multiple masks can be created to refer to the same bit flag.

The resulting mask isolates only one bit flag in the BitVector32. You can combine masks using the bitwise OR operation to create a mask that isolates multiple bit flags in the BitVector32.

Using a mask on a BitVector32 that is set up as sections might cause unexpected results.

This method is an O(1) operation.

BitVector32.CreateSection BitVector32.CreateSection

In this Article

Overloads

CreateSection(Int16) CreateSection(Int16)	Creates the first BitVector32.Section in a series of sections that contain small integers.
CreateSection(Int16, BitVector32+Section) CreateSection(Int16, BitVector32+Section)	Creates a new BitVector32.Section following the specified BitVector32.Section in a series of sections that contain small integers.

CreateSection(Int16) CreateSection(Int16)

Creates the first BitVector32.Section in a series of sections that contain small integers.

```
public static System.Collections.Specialized.BitVector32.Section CreateSection (short maxValue);
```

```
static member CreateSection : int16 -> System.Collections.Specialized.BitVector32.Section
```

Parameters

maxValue Int16 Int16

A 16-bit signed integer that specifies the maximum value for the new BitVector32.Section.

Returns

BitVector32.Section BitVector32.Section

A BitVector32.Section that can hold a number from zero to `maxValue`.

Exceptions

ArgumentException ArgumentException

`maxValue` is less than 1.

Examples

The following code example uses a BitVector32 as a collection of sections.

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


public class SamplesBitVector32  {

   public static void Main()  {

      // Creates and initializes a BitVector32.
      BitVector32 myBV = new BitVector32( 0 );

      // Creates four sections in the BitVector32 with maximum values 6, 3, 1, and 15.
      // mySect3, which uses exactly one bit, can also be used as a bit flag.
      BitVector32.Section mySect1 = BitVector32.CreateSection( 6 );
      BitVector32.Section mySect2 = BitVector32.CreateSection( 3, mySect1 );
      BitVector32.Section mySect3 = BitVector32.CreateSection( 1, mySect2 );
      BitVector32.Section mySect4 = BitVector32.CreateSection( 15, mySect3 );
```

```
      BitVector32.Section mySect4 = BitVector32.CreateSection( 15, mySect3 );

      // Displays the values of the sections.
      Console.WriteLine( "Initial values:" );
      Console.WriteLine( "    mySect1: {0}", myBV[mySect1] );
      Console.WriteLine( "    mySect2: {0}", myBV[mySect2] );
      Console.WriteLine( "    mySect3: {0}", myBV[mySect3] );
      Console.WriteLine( "    mySect4: {0}", myBV[mySect4] );

      // Sets each section to a new value and displays the value of the BitVector32 at each step.
      Console.WriteLine( "Changing the values of each section:" );
      Console.WriteLine( "    Initial:        {0}", myBV.ToString() );
      myBV[mySect1] = 5;
      Console.WriteLine( "    mySect1 = 5:    {0}", myBV.ToString() );
      myBV[mySect2] = 3;
      Console.WriteLine( "    mySect2 = 3:    {0}", myBV.ToString() );
      myBV[mySect3] = 1;
      Console.WriteLine( "    mySect3 = 1:    {0}", myBV.ToString() );
      myBV[mySect4] = 9;
      Console.WriteLine( "    mySect4 = 9:    {0}", myBV.ToString() );

      // Displays the values of the sections.
      Console.WriteLine( "New values:" );
      Console.WriteLine( "    mySect1: {0}", myBV[mySect1] );
      Console.WriteLine( "    mySect2: {0}", myBV[mySect2] );
      Console.WriteLine( "    mySect3: {0}", myBV[mySect3] );
      Console.WriteLine( "    mySect4: {0}", myBV[mySect4] );

   }

}

/*
This code produces the following output.

Initial values:
        mySect1: 0
        mySect2: 0
        mySect3: 0
        mySect4: 0
Changing the values of each section:
        Initial:        BitVector32{00000000000000000000000000000000}
        mySect1 = 5:    BitVector32{00000000000000000000000000000101}
        mySect2 = 3:    BitVector32{00000000000000000000000000011101}
        mySect3 = 1:    BitVector32{00000000000000000000000000111101}
        mySect4 = 9:    BitVector32{00000000000000000000001001111101}
New values:
        mySect1: 5
        mySect2: 3
        mySect3: 1
        mySect4: 9

*/
```

Remarks

A BitVector32.Section is a window into the BitVector32 and is composed of the smallest number of consecutive bits that can contain the maximum value specified in CreateSection. For example, a section with a maximum value of 1 is composed of only one bit, whereas a section with a maximum value of 5 is composed of three bits. You can create a BitVector32.Section with a maximum value of 1 to serve as a Boolean, thereby allowing you to store integers and Booleans in the same BitVector32.

If sections already exist in the BitVector32, those sections are still accessible; however, overlapping sections might cause unexpected results.

This method is an O(1) operation.

CreateSection(Int16, BitVector32+Section) CreateSection(Int16, BitVector32+Section)

Creates a new BitVector32.Section following the specified BitVector32.Section in a series of sections that contain small integers.

```
public static System.Collections.Specialized.BitVector32.Section CreateSection (short maxValue,
System.Collections.Specialized.BitVector32.Section previous);
```

```
static member CreateSection : int16 * System.Collections.Specialized.BitVector32.Section ->
System.Collections.Specialized.BitVector32.Section
```

Parameters

maxValue Int16 Int16

A 16-bit signed integer that specifies the maximum value for the new BitVector32.Section.

previous BitVector32.Section BitVector32.Section

The previous BitVector32.Section in the BitVector32.

Returns
BitVector32.Section BitVector32.Section

A BitVector32.Section that can hold a number from zero to `maxValue`.

Exceptions
ArgumentException ArgumentException

`maxValue` is less than 1.

InvalidOperationException InvalidOperationException

`previous` includes the final bit in the BitVector32.

-or-

`maxValue` is greater than the highest value that can be represented by the number of bits after `previous`.

Examples

The following code example uses a BitVector32 as a collection of sections.

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


public class SamplesBitVector32  {

   public static void Main()  {

      // Creates and initializes a BitVector32.
      BitVector32 myBV = new BitVector32( 0 );

      // Creates four sections in the BitVector32 with maximum values 6, 3, 1, and 15.
      // mySect3, which uses exactly one bit, can also be used as a bit flag.
      BitVector32.Section mySect1 = BitVector32.CreateSection( 6 );
      BitVector32.Section mySect2 = BitVector32.CreateSection( 3, mySect1 );
      BitVector32.Section mySect3 = BitVector32.CreateSection( 1, mySect2 );
```

```
      BitVector32.Section mySect4 = BitVector32.CreateSection( 15, mySect3 );

      // Displays the values of the sections.
      Console.WriteLine( "Initial values:" );
      Console.WriteLine( "    mySect1: {0}", myBV[mySect1] );
      Console.WriteLine( "    mySect2: {0}", myBV[mySect2] );
      Console.WriteLine( "    mySect3: {0}", myBV[mySect3] );
      Console.WriteLine( "    mySect4: {0}", myBV[mySect4] );

      // Sets each section to a new value and displays the value of the BitVector32 at each step.
      Console.WriteLine( "Changing the values of each section:" );
      Console.WriteLine( "    Initial:        {0}", myBV.ToString() );
      myBV[mySect1] = 5;
      Console.WriteLine( "    mySect1 = 5:    {0}", myBV.ToString() );
      myBV[mySect2] = 3;
      Console.WriteLine( "    mySect2 = 3:    {0}", myBV.ToString() );
      myBV[mySect3] = 1;
      Console.WriteLine( "    mySect3 = 1:    {0}", myBV.ToString() );
      myBV[mySect4] = 9;
      Console.WriteLine( "    mySect4 = 9:    {0}", myBV.ToString() );

      // Displays the values of the sections.
      Console.WriteLine( "New values:" );
      Console.WriteLine( "    mySect1: {0}", myBV[mySect1] );
      Console.WriteLine( "    mySect2: {0}", myBV[mySect2] );
      Console.WriteLine( "    mySect3: {0}", myBV[mySect3] );
      Console.WriteLine( "    mySect4: {0}", myBV[mySect4] );

   }

}

/*
This code produces the following output.

Initial values:
        mySect1: 0
        mySect2: 0
        mySect3: 0
        mySect4: 0
Changing the values of each section:
        Initial:        BitVector32{00000000000000000000000000000000}
        mySect1 = 5:    BitVector32{00000000000000000000000000000101}
        mySect2 = 3:    BitVector32{00000000000000000000000000011101}
        mySect3 = 1:    BitVector32{00000000000000000000000000111101}
        mySect4 = 9:    BitVector32{00000000000000000000001001111101}
New values:
        mySect1: 5
        mySect2: 3
        mySect3: 1
        mySect4: 9

*/
```

Remarks

A BitVector32.Section is a window into the BitVector32 and is composed of the smallest number of consecutive bits that can contain the maximum value specified in CreateSection. For example, a section with a maximum value of 1 is composed of only one bit, whereas a section with a maximum value of 5 is composed of three bits. You can create a BitVector32.Section with a maximum value of 1 to serve as a Boolean, thereby allowing you to store integers and Booleans in the same BitVector32.

If sections already exist after `previous` in the BitVector32, those sections are still accessible; however, overlapping sections might cause unexpected results.

This method is an O(1) operation.

BitVector32.Data BitVector32.Data

In this Article

Gets the value of the BitVector32 as an integer.

```
public int Data { get; }
```

```
member this.Data : int
```

Returns

Int32 Int32

The value of the BitVector32 as an integer.

Remarks

To access the value of the individual sections or bit flags, use the Item[Int32] property.

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

See Also

Item[Int32]Item[Int32]

BitVector32.Equals BitVector32.Equals

In this Article

Determines whether the specified object is equal to the BitVector32.

```
public override bool Equals (object o);
```

```
override this.Equals : obj -> bool
```

Parameters

o Object Object

The object to compare with the current BitVector32.

Returns

Boolean Boolean

`true` if the specified object is equal to the BitVector32; otherwise, `false`.

Examples

The following code example compares a BitVector32 with another BitVector32 and with an Int32.

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


public class SamplesBitVector32  {

   public static void Main()  {

      // Creates and initializes a BitVector32 with the value 123.
      // This is the BitVector32 that will be compared to different types.
      BitVector32 myBV = new BitVector32( 123 );

      // Creates and initializes a new BitVector32 which will be set up as sections.
      BitVector32 myBVsect = new BitVector32( 0 );

      // Compares myBV and myBVsect.
      Console.WriteLine( "myBV                 : {0}", myBV.ToString() );
      Console.WriteLine( "myBVsect             : {0}", myBVsect.ToString() );
      if ( myBV.Equals( myBVsect ) )
         Console.WriteLine( "   myBV({0}) equals myBVsect({1}).", myBV.Data, myBVsect.Data );
      else
         Console.WriteLine( "   myBV({0}) does not equal myBVsect({1}).", myBV.Data, myBVsect.Data
);
      Console.WriteLine();

      // Assigns values to the sections of myBVsect.
      BitVector32.Section mySect1 = BitVector32.CreateSection( 5 );
      BitVector32.Section mySect2 = BitVector32.CreateSection( 1, mySect1 );
      BitVector32.Section mySect3 = BitVector32.CreateSection( 20, mySect2 );
      myBVsect[mySect1] = 3;
      myBVsect[mySect2] = 1;
      myBVsect[mySect3] = 7;

      // Compares myBV and myBVsect.
      Console.WriteLine( "myBV                 : {0}", myBV.ToString() );
      Console.WriteLine( "myBVsect with values : {0}", myBVsect.ToString() );
      if ( myBV.Equals( myBVsect ) )
         Console.WriteLine( "   myBV({0}) equals myBVsect({1}).", myBV.Data, myBVsect.Data );
```

```
      else
         Console.WriteLine( "   myBV({0}) does not equal myBVsect({1}).", myBV.Data, myBVsect.Data
);
      Console.WriteLine();

      // Compare myBV with an Int32.
      Console.WriteLine( "Comparing myBV with an Int32: " );
      Int32 myInt32 = 123;
      // Using Equals will fail because Int32 is not compatible with BitVector32.
      if ( myBV.Equals( myInt32 ) )
         Console.WriteLine( "   Using BitVector32.Equals, myBV({0}) equals myInt32({1}).",
myBV.Data, myInt32 );
      else
         Console.WriteLine( "   Using BitVector32.Equals, myBV({0}) does not equal myInt32({1}).",
myBV.Data, myInt32 );
      // To compare a BitVector32 with an Int32, use the "==" operator.
      if ( myBV.Data == myInt32 )
         Console.WriteLine( "   Using the \"==\" operator, myBV.Data({0}) equals myInt32({1}).",
myBV.Data, myInt32 );
      else
         Console.WriteLine( "   Using the \"==\" operator, myBV.Data({0}) does not equal
myInt32({1}).", myBV.Data, myInt32 );

   }

}

/*
This code produces the following output.

myBV                 : BitVector32{00000000000000000000000001111011}
myBVsect             : BitVector32{00000000000000000000000000000000}
   myBV(123) does not equal myBVsect(0).

myBV                 : BitVector32{00000000000000000000000001111011}
myBVsect with values : BitVector32{00000000000000000000000001111011}
   myBV(123) equals myBVsect(123).

Comparing myBV with an Int32:
   Using BitVector32.Equals, myBV(123) does not equal myInt32(123).
   Using the "==" operator, myBV.Data(123) equals myInt32(123).

*/
```

Remarks

The object `o` is considered equal to the BitVector32 if the type of `o` is compatible with the BitVector32 type and if the value of `o` is equal to the value of Data.

This method is an O(1) operation.

BitVector32.GetHashCode BitVector32.GetHashCode

In this Article

Serves as a hash function for the BitVector32.

```
public override int GetHashCode ();
```

```
override this.GetHashCode : unit -> int
```

Returns

Int32 Int32

A hash code for the BitVector32.

Remarks

The hash code of a BitVector32 is based on the value of Data. Two instances of BitVector32 with the same value for Data will also generate the same hash code.

This method is an O(1) operation.

BitVector32.Item[Int32] BitVector32.Item[Int32]

In this Article

Overloads

Item[BitVector32+Section] Item[BitVector32+Section]	Gets or sets the value stored in the specified BitVector32.Section.
Item[Int32] Item[Int32]	Gets or sets the state of the bit flag indicated by the specified mask.

Item[BitVector32+Section] Item[BitVector32+Section]

Gets or sets the value stored in the specified BitVector32.Section.

```
public int this[System.Collections.Specialized.BitVector32.Section section] { get; set; }
```

```
member this.Item(System.Collections.Specialized.BitVector32.Section) : int with get, set
```

Parameters

section BitVector32.Section BitVector32.Section

A BitVector32.Section that contains the value to get or set.

Returns
Int32 Int32

The value stored in the specified BitVector32.Section.

Remarks

The Item[Int32] [Section] property is the indexer for a BitVector32 that is set up as sections, and the Item[Int32] [int] property is the indexer for a BitVector32 that is set up as bit flags.

A BitVector32.Section is a window into the BitVector32 and is composed of the smallest number of consecutive bits that can contain the maximum value specified in CreateSection. For example, a section with a maximum value of 1 is composed of only one bit, whereas a section with a maximum value of 5 is composed of three bits. You can create a BitVector32.Section with a maximum value of 1 to serve as a Boolean, thereby allowing you to store integers and Booleans in the same BitVector32.

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 Also
CreateSection(Int16)CreateSection(Int16)
BitVector32.SectionBitVector32.Section

Item[Int32] Item[Int32]

Gets or sets the state of the bit flag indicated by the specified mask.

```
public bool this[int bit] { get; set; }
```

```
member this.Item(int) : bool with get, set
```

Parameters

bit Int32 Int32

A mask that indicates the bit to get or set.

Returns
Boolean Boolean

`true` if the specified bit flag is on (1); otherwise, `false`.

Remarks

The Item[Int32] [Section] property is the indexer for a BitVector32 that is set up as sections, and the Item[Int32] [int] property is the indexer for a BitVector32 that is set up as bit flags.

Using this property on a BitVector32 that is set up as sections might cause unexpected results.

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 Also CreateMask()CreateMask()

BitVector32.ToString BitVector32.ToString

In this Article

Overloads

ToString() ToString()	Returns a string that represents the current BitVector32.
ToString(BitVector32) ToString(BitVector32)	Returns a string that represents the specified BitVector32.

ToString() ToString()

Returns a string that represents the current BitVector32.

```
public override string ToString ();
```

```
override this.ToString : unit -> string
```

Returns

String String

A string that represents the current BitVector32.

Remarks

This method overrides Object.ToString.

This method is an O(1) operation.

See Also

StringString

ToString(BitVector32) ToString(BitVector32)

Returns a string that represents the specified BitVector32.

```
public static string ToString (System.Collections.Specialized.BitVector32 value);
```

```
static member ToString : System.Collections.Specialized.BitVector32 -> string
```

Parameters

value

BitVector32 BitVector32

The BitVector32 to represent.

Returns

String String

A string that represents the specified BitVector32.

Remarks

This method is an O(1) operation.

See

StringString

Also

BitVector32.Section BitVector32.Section Struct

Represents a section of the vector that can contain an integer number.

Declaration

```
public struct BitVector32.Section
```

```
type BitVector32.Section = struct
```

Inheritance Hierarchy

`Object` `Object`

`ValueType` `ValueType`

Remarks

Use CreateSection to define a new section. A BitVector32.Section is a window into the BitVector32 and is composed of the smallest number of consecutive bits that can contain the maximum value specified in CreateSection. For example, a section with a maximum value of 1 is composed of only one bit, whereas a section with a maximum value of 5 is composed of three bits. You can create a BitVector32.Section with a maximum value of 1 to serve as a Boolean, thereby allowing you to store integers and Booleans in the same BitVector32.

Properties

`Mask`

`Mask`

Gets a mask that isolates this section within the BitVector32.

`Offset`

`Offset`

Gets the offset of this section from the start of the BitVector32.

Methods

`Equals(BitVector32+Section)`

`Equals(BitVector32+Section)`

Determines whether the specified BitVector32.Section object is the same as the current BitVector32.Section object.

`Equals(Object)`

`Equals(Object)`

Determines whether the specified object is the same as the current BitVector32.Section object.

`GetHashCode()`

`GetHashCode()`

Serves as a hash function for the current BitVector32.Section, suitable for hashing algorithms and data structures, such as a hash table.

`ToString()`

`ToString()`

Returns a string that represents the current BitVector32.Section.

`ToString(BitVector32+Section)`

`ToString(BitVector32+Section)`

Returns a string that represents the specified BitVector32.Section.

Operators

`Equality(BitVector32+Section, BitVector32+Section)`

`Equality(BitVector32+Section, BitVector32+Section)`

Determines whether two specified BitVector32.Section objects are equal.

`Inequality(BitVector32+Section, BitVector32+Section)`

`Inequality(BitVector32+Section, BitVector32+Section)`

Determines whether two BitVector32.Section objects have different values.

BitVector32.Section.Equality BitVector32.Section.Equality

In this Article

Determines whether two specified BitVector32.Section objects are equal.

```
public static bool operator == (System.Collections.Specialized.BitVector32.Section a,
System.Collections.Specialized.BitVector32.Section b);
```

```
static member ( = ) : System.Collections.Specialized.BitVector32.Section *
System.Collections.Specialized.BitVector32.Section -> bool
```

Parameters

a BitVector32.Section BitVector32.Section

A BitVector32.Section object.

b BitVector32.Section BitVector32.Section

A BitVector32.Section object.

Returns

Boolean Boolean

`true` if the `a` and `b` parameters represent the same BitVector32.Section object, otherwise, `false`.

Remarks

The equivalent method for this operator is BitVector32.Section.Equals(BitVector32+Section).

BitVector32.Section.Equals BitVector32.Section.Equals

In this Article

Overloads

Equals(BitVector32+Section) Equals(BitVector32+Section)	Determines whether the specified BitVector32.Section object is the same as the current BitVector32.Section object.
Equals(Object) Equals(Object)	Determines whether the specified object is the same as the current BitVector32.Section object.

Equals(BitVector32+Section) Equals(BitVector32+Section)

Determines whether the specified BitVector32.Section object is the same as the current BitVector32.Section object.

```
public bool Equals (System.Collections.Specialized.BitVector32.Section obj);
```

```
override this.Equals : System.Collections.Specialized.BitVector32.Section -> bool
```

Parameters

obj BitVector32.Section BitVector32.Section

The BitVector32.Section object to compare with the current BitVector32.Section object.

Returns

Boolean Boolean

`true` if the `obj` parameter is the same as the current BitVector32.Section object; otherwise `false`.

Equals(Object) Equals(Object)

Determines whether the specified object is the same as the current BitVector32.Section object.

```
public override bool Equals (object o);
```

```
override this.Equals : obj -> bool
```

Parameters

o Object Object

The object to compare with the current BitVector32.Section.

Returns

Boolean Boolean

`true` if the specified object is the same as the current BitVector32.Section object; otherwise, `false`.

Remarks

This method overrides Object.Equals.

Two BitVector32.Section instances are considered equal if both sections are of the same length and are in the same location within a BitVector32.

BitVector32.Section.GetHashCode BitVector32.Section.GetHashCode

In this Article

Serves as a hash function for the current BitVector32.Section, suitable for hashing algorithms and data structures, such as a hash table.

```
public override int GetHashCode ();
```

```
override this.GetHashCode : unit -> int
```

Returns

Int32 Int32

A hash code for the current BitVector32.Section.

Remarks

This method overrides Object.GetHashCode.

This method generates the same hash code for two objects that are equal according to the Equals method.

See Also

Equals(Object)Equals(Object)

BitVector32.Section.Inequality BitVector32.Section.Inequality

In this Article

Determines whether two BitVector32.Section objects have different values.

```
public static bool operator != (System.Collections.Specialized.BitVector32.Section a,
System.Collections.Specialized.BitVector32.Section b);
```

```
static member op_Inequality : System.Collections.Specialized.BitVector32.Section *
System.Collections.Specialized.BitVector32.Section -> bool
```

Parameters

a BitVector32.Section BitVector32.Section

A BitVector32.Section object.

b BitVector32.Section BitVector32.Section

A BitVector32.Section object.

Returns
Boolean Boolean

`true` if the `a` and `b` parameters represent different BitVector32.Section objects; otherwise, `false`.

Remarks

The equivalent method for this operator is Equality Method System.Collections.Specialized 4.0.1.0 4.1.0.0 System 2.0.5.0 4.0.0.0 netstandard 2.0.0.0 System.String

Returns a string that represents the current .
A string that represents the current . <![CDATA[

This method overrides Object.ToString.

BitVector32.Section.Mask BitVector32.Section.Mask

In this Article

Gets a mask that isolates this section within the BitVector32.

```
public short Mask { get; }
```

```
member this.Mask : int16
```

Returns

Int16 Int16

A mask that isolates this section within the BitVector32.

BitVector32.Section.Offset BitVector32.Section.Offset

In this Article

Gets the offset of this section from the start of the BitVector32.

```
public short Offset { get; }
```

```
member this.Offset : int16
```

Returns

Int16 Int16

The offset of this section from the start of the BitVector32.

BitVector32.Section.ToString BitVector32.Section.ToString

In this Article

Overloads

ToString() ToString()	Returns a string that represents the current BitVector32.Section.
ToString(BitVector32+Section) ToString(BitVector32+Section)	Returns a string that represents the specified BitVector32.Section.

ToString() ToString()

Returns a string that represents the current BitVector32.Section.

```
public override string ToString ();
```

```
override this.ToString : unit -> string
```

Returns

String String

A string that represents the current BitVector32.Section.

Remarks

This method overrides Object.ToString.

See Also

StringString

ToString(BitVector32+Section) ToString(BitVector32+Section)

Returns a string that represents the specified BitVector32.Section.

```
public static string ToString (System.Collections.Specialized.BitVector32.Section value);
```

```
static member ToString : System.Collections.Specialized.BitVector32.Section -> string
```

Parameters

value

BitVector32.Section BitVector32.Section

The BitVector32.Section to represent.

Returns

String String

A string that represents the specified BitVector32.Section.

See Also

StringString

CollectionChangedEventManager CollectionChanged EventManager Class

Provides a WeakEventManager implementation so that you can use the "weak event listener" pattern to attach listeners for the CollectionChanged event.

Declaration

```
public class CollectionChangedEventManager : System.Windows.WeakEventManager
```

```
type CollectionChangedEventManager = class
    inherit WeakEventManager
```

Inheritance Hierarchy

`Object` `Object`
`DispatcherObject` `DispatcherObject`
`WeakEventManager` `WeakEventManager`

Remarks

In order to be listeners in this pattern, your listener objects must implement IWeakEventListener. You do not need to implement IWeakEventListener on the class that is the source of the events.

Methods

`AddHandler(INotifyCollectionChanged, EventHandler<NotifyCollectionChangedEventArgs>)`

`AddHandler(INotifyCollectionChanged, EventHandler<NotifyCollectionChangedEventArgs>)`

Adds the specified event handler, which is called when specified source raises the CollectionChanged event.

`AddListener(INotifyCollectionChanged, IWeakEventListener)`

`AddListener(INotifyCollectionChanged, IWeakEventListener)`

Adds the specified listener to the CollectionChanged event of the specified source.

`NewListenerList()`

`NewListenerList()`

Returns a new object to contain listeners to the CollectionChanged event.

`RemoveHandler(INotifyCollectionChanged, EventHandler<NotifyCollectionChangedEventArgs>)`

`RemoveHandler(INotifyCollectionChanged, EventHandler<NotifyCollectionChangedEventArgs>)`

Removes the specified event handler from the specified source.

`RemoveListener(INotifyCollectionChanged, IWeakEventListener)`

`RemoveListener(INotifyCollectionChanged, IWeakEventListener)`

Removes the specified listener from the CollectionChanged event of the specified source.

`StartListening(Object)`

`StartListening(Object)`

Begins listening for the CollectionChanged event on the specified source.

`StopListening(Object)`

`StopListening(Object)`

Stops listening for the CollectionChanged event on the specified source.

See Also

WeakEventManager WeakEventManager

CollectionChangedEventManager.AddHandler CollectionChangedEventManager.AddHandler

In this Article

Adds the specified event handler, which is called when specified source raises the CollectionChanged event.

```
public static void AddHandler (System.Collections.Specialized.INotifyCollectionChanged source,
EventHandler<System.Collections.Specialized.NotifyCollectionChangedEventArgs> handler);
```

```
static member AddHandler : System.Collections.Specialized.INotifyCollectionChanged *
EventHandler<System.Collections.Specialized.NotifyCollectionChangedEventArgs> -> unit
```

Parameters

source INotifyCollectionChanged INotifyCollectionChanged

The source object that the raises the CollectionChanged event.

handler EventHandler<NotifyCollectionChangedEventArgs>

The delegate that handles the CollectionChanged event.

CollectionChangedEventManager.AddListener CollectionChangedEventManager.AddListener

In this Article

Adds the specified listener to the CollectionChanged event of the specified source.

```
public static void AddListener (System.Collections.Specialized.INotifyCollectionChanged source,
System.Windows.IWeakEventListener listener);
```

```
static member AddListener : System.Collections.Specialized.INotifyCollectionChanged *
System.Windows.IWeakEventListener -> unit
```

Parameters

source INotifyCollectionChanged INotifyCollectionChanged

The object with the event.

listener IWeakEventListener IWeakEventListener

The object to add as a listener.

CollectionChangedEventManager.NewListenerList CollectionChangedEventManager.NewListenerList

In this Article

Returns a new object to contain listeners to the CollectionChanged event.

```
protected override System.Windows.WeakEventManager.ListenerList NewListenerList ();
```

```
override this.NewListenerList : unit -> System.Windows.WeakEventManager.ListenerList
```

Returns
WeakEventManager.ListenerList WeakEventManager.ListenerList

A new object to contain listeners to the CollectionChanged event.

CollectionChangedEventManager.RemoveHandler CollectionChangedEventManager.RemoveHandler

In this Article

Removes the specified event handler from the specified source.

```
public static void RemoveHandler (System.Collections.Specialized.INotifyCollectionChanged source,
EventHandler<System.Collections.Specialized.NotifyCollectionChangedEventArgs> handler);
```

```
static member RemoveHandler : System.Collections.Specialized.INotifyCollectionChanged *
EventHandler<System.Collections.Specialized.NotifyCollectionChangedEventArgs> -> unit
```

Parameters

source INotifyCollectionChanged INotifyCollectionChanged

The source object that the raises the CollectionChanged event.

handler EventHandler<NotifyCollectionChangedEventArgs>

The delegate that handles the CollectionChanged event.

CollectionChangedEventManager.RemoveListener CollectionChangedEventManager.RemoveListener

In this Article

Removes the specified listener from the CollectionChanged event of the specified source.

```
public static void RemoveListener (System.Collections.Specialized.INotifyCollectionChanged source,
System.Windows.IWeakEventListener listener);
```

```
static member RemoveListener : System.Collections.Specialized.INotifyCollectionChanged *
System.Windows.IWeakEventListener -> unit
```

Parameters

source INotifyCollectionChanged INotifyCollectionChanged

The object with the event.

listener IWeakEventListener IWeakEventListener

The listener to remove.

CollectionChangedEventManager.StartListening CollectionChangedEventManager.StartListening

In this Article

Begins listening for the CollectionChanged event on the specified source.

```
protected override void StartListening (object source);
```

```
override this.StartListening : obj -> unit
```

Parameters

source Object Object

The object with the event.

CollectionChangedEventManager.StopListening CollectionChangedEventManager.StopListening

In this Article

Stops listening for the CollectionChanged event on the specified source.

```
protected override void StopListening (object source);
```

```
override this.StopListening : obj -> unit
```

Parameters

source Object Object

The object with the event.

CollectionsUtil CollectionsUtil Class

Creates collections that ignore the case in strings.

Declaration

```
public class CollectionsUtil
```

```
type CollectionsUtil = class
```

Inheritance Hierarchy

`Object` `Object`

Remarks

These methods generate a case-insensitive instance of the collection using case-insensitive implementations of the hash code provider and the comparer. The resulting instance can be used like any other instances of that class, although it may behave differently.

For example, suppose two objects with the keys "hello" and "HELLO" are to be added to a hash table. A case-sensitive hash table would create two different entries; whereas, a case-insensitive hash table would throw an exception when adding the second object.

Constructors

`CollectionsUtil()`

`CollectionsUtil()`

Initializes a new instance of the CollectionsUtil class.

Methods

`CreateCaseInsensitiveHashtable()`

`CreateCaseInsensitiveHashtable()`

Creates a new case-insensitive instance of the Hashtable class with the default initial capacity.

`CreateCaseInsensitiveHashtable(IDictionary)`

`CreateCaseInsensitiveHashtable(IDictionary)`

Copies the entries from the specified dictionary to a new case-insensitive instance of the Hashtable class with the same initial capacity as the number of entries copied.

`CreateCaseInsensitiveHashtable(Int32)`

`CreateCaseInsensitiveHashtable(Int32)`

Creates a new case-insensitive instance of the Hashtable class with the specified initial capacity.

`CreateCaseInsensitiveSortedList()`

`CreateCaseInsensitiveSortedList()`

Creates a new instance of the SortedList class that ignores the case of strings.

Thread Safety

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(Hashtable) method.

A SortedList 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

CollectionsUtil

In this Article

Initializes a new instance of the CollectionsUtil class.

```
public CollectionsUtil ();
```

CollectionsUtil.CreateCaseInsensitiveHashtable CollectionsUtil.CreateCaseInsensitiveHashtable

In this Article

Overloads

CreateCaseInsensitiveHashtable() CreateCaseInsensitive Hashtable()	Creates a new case-insensitive instance of the Hashtable class with the default initial capacity.
CreateCaseInsensitiveHashtable(IDictionary) CreateCase InsensitiveHashtable(IDictionary)	Copies the entries from the specified dictionary to a new case-insensitive instance of the Hashtable class with the same initial capacity as the number of entries copied.
CreateCaseInsensitiveHashtable(Int32) CreateCaseInsensitive Hashtable(Int32)	Creates a new case-insensitive instance of the Hashtable class with the specified initial capacity.

CreateCaseInsensitiveHashtable() CreateCaseInsensitiveHashtable()

Creates a new case-insensitive instance of the Hashtable class with the default initial capacity.

```
public static System.Collections.Hashtable CreateCaseInsensitiveHashtable ();
```

```
static member CreateCaseInsensitiveHashtable : unit -> System.Collections.Hashtable
```

Returns
Hashtable Hashtable

A new case-insensitive instance of the Hashtable class with the default initial capacity.

Remarks

Instead of using the CreateCaseInsensitiveHashtable method, use the Hashtable.Hashtable(IEqualityComparer) constructor to create a case-insensitive Hashtable class.

See Also	HashtableHashtable Performing Culture-Insensitive String Operations in Collections

CreateCaseInsensitiveHashtable(IDictionary) CreateCaseInsensitiveHashtable(IDictionary)

Copies the entries from the specified dictionary to a new case-insensitive instance of the Hashtable class with the same initial capacity as the number of entries copied.

```
public static System.Collections.Hashtable CreateCaseInsensitiveHashtable
(System.Collections.IDictionary d);
```

```
static member CreateCaseInsensitiveHashtable : System.Collections.IDictionary ->
System.Collections.Hashtable
```

Parameters

d IDictionary IDictionary

The IDictionary to copy to a new case-insensitive Hashtable.

Returns
Hashtable Hashtable

A new case-insensitive instance of the Hashtable class containing the entries from the specified IDictionary.

Exceptions
ArgumentNullException ArgumentNullException

`d` is `null`.

Remarks

Instead of using the CreateCaseInsensitiveHashtable method, use the Hashtable.Hashtable(IDictionary, IEqualityComparer) constructor to create a case-insensitive Hashtable class.

See Also
HashtableHashtable
IDictionaryIDictionary
Performing Culture-Insensitive String Operations in Collections

CreateCaseInsensitiveHashtable(Int32) CreateCaseInsensitiveHashtable(Int32)

Creates a new case-insensitive instance of the Hashtable class with the specified initial capacity.

```
public static System.Collections.Hashtable CreateCaseInsensitiveHashtable (int capacity);
```

```
static member CreateCaseInsensitiveHashtable : int -> System.Collections.Hashtable
```

Parameters

capacity Int32 Int32

The approximate number of entries that the Hashtable can initially contain.

Returns
Hashtable Hashtable

A new case-insensitive instance of the Hashtable class with the specified initial capacity.

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

Remarks

Instead of using the CreateCaseInsensitiveHashtable method, use the Hashtable.Hashtable(Int32, IEqualityComparer) constructor to create a case-insensitive Hashtable class.

See Also
HashtableHashtable
Performing Culture-Insensitive String Operations in Collections

CollectionsUtil.CreateCaseInsensitiveSortedList CollectionsUtil.CreateCaseInsensitiveSortedList

In this Article

Creates a new instance of the SortedList class that ignores the case of strings.

```
public static System.Collections.SortedList CreateCaseInsensitiveSortedList ();
```

```
static member CreateCaseInsensitiveSortedList : unit -> System.Collections.SortedList
```

Returns

SortedList SortedList

A new instance of the SortedList class that ignores the case of strings.

Remarks

The new SortedList instance is sorted according to the CaseInsensitiveComparer.

See Also	SortedListSortedList Performing Culture-Insensitive String Operations in Collections

HybridDictionary HybridDictionary Class

Implements `IDictionary` by using a ListDictionary while the collection is small, and then switching to a Hashtable when the collection gets large.

Declaration

```
[Serializable]
public class HybridDictionary : System.Collections.IDictionary
```

```
type HybridDictionary = class
    interface IDictionary
    interface ICollection
    interface IEnumerable
```

Inheritance Hierarchy

`Object` `Object`

Remarks

This class is recommended for cases where the number of elements in a dictionary is unknown. It takes advantage of the improved performance of a ListDictionary with small collections, and offers the flexibility of switching to a Hashtable which handles larger collections better than ListDictionary.

If the initial size of the collection is greater than the optimal size for a ListDictionary, the collection is stored in a Hashtable to avoid the overhead of copying elements from the ListDictionary to a Hashtable.

The constructor accepts a Boolean parameter that allows the user to specify whether the collection ignores the case when comparing strings. If the collection is case-sensitive, it uses the key's implementations of Object.GetHashCode and Object.Equals. If the collection is case-insensitive, it performs a simple ordinal case-insensitive comparison, which obeys the casing rules of the invariant culture only. By default, the collection is case-sensitive. For more information on the invariant culture, see System.Globalization.CultureInfo.

A key cannot be null, but a value can.

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 HybridDictionary 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 myHybridDictionary)
{
    //...
}
```

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

Constructors

`HybridDictionary()`

`HybridDictionary()`

Creates an empty case-sensitive HybridDictionary.

`HybridDictionary(Boolean)`

`HybridDictionary(Boolean)`

Creates an empty HybridDictionary with the specified case sensitivity.

`HybridDictionary(Int32)`

`HybridDictionary(Int32)`

Creates a case-sensitive HybridDictionary with the specified initial size.

`HybridDictionary(Int32, Boolean)`

`HybridDictionary(Int32, Boolean)`

Creates a HybridDictionary with the specified initial size and case sensitivity.

Properties

`Count`

`Count`

Gets the number of key/value pairs contained in the HybridDictionary.

`IsFixedSize`

`IsFixedSize`

Gets a value indicating whether the HybridDictionary has a fixed size.

`IsReadOnly`

`IsReadOnly`

Gets a value indicating whether the HybridDictionary is read-only.

`IsSynchronized`

`IsSynchronized`

Gets a value indicating whether the HybridDictionary 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 HybridDictionary.

SyncRoot

SyncRoot

Gets an object that can be used to synchronize access to the HybridDictionary.

Values

Values

Gets an ICollection containing the values in the HybridDictionary.

Methods

Add(Object, Object)

Add(Object, Object)

Adds an entry with the specified key and value into the HybridDictionary.

Clear()

Clear()

Removes all entries from the HybridDictionary.

Contains(Object)

Contains(Object)

Determines whether the HybridDictionary contains a specific key.

CopyTo(Array, Int32)

CopyTo(Array, Int32)

Copies the HybridDictionary entries to a one-dimensional Array instance at the specified index.

GetEnumerator()

GetEnumerator()

Returns an IDictionaryEnumerator that iterates through the HybridDictionary.

Remove(Object)

Remove(Object)

Removes the entry with the specified key from the HybridDictionary.

IEnumerable.GetEnumerator()

IEnumerable.GetEnumerator()

Returns an IEnumerator that iterates through the HybridDictionary.

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 HybridDictionary, but derived classes can create their own synchronized versions of the HybridDictionary 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

IDictionary IDictionary
GetHashCode() GetHashCode()
Equals(Object) Equals(Object)
IDictionary IDictionary

HybridDictionary.Add HybridDictionary.Add

In this Article

Adds an entry with the specified key and value into the HybridDictionary.

```
public 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 entry to add.

value Object Object

The value of the entry to add. The value can be `null`.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

ArgumentException ArgumentException

An entry with the same key already exists in the HybridDictionary.

Examples

The following code example adds to and removes elements from a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );
```

```
      // Displays the values in the HybridDictionary in three different ways.
      Console.WriteLine( "Initial contents of the HybridDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":"
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                        VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the HybridDictionary:
   KEY                        VALUE
   Seedless Watermelon        0.49
   Nectarine                  1.99
   Cranberries                5.98
   Plantain Bananas           1.49
   Honeydew Melon             0.59
   Pineapple                  1.49
   Strawberries               3.33
   Grapes                     1.99
   Braeburn Apples            1.49
   Peaches                    1.99
   Red Delicious Apples       0.99
   Golden Delicious Apples    1.29
   Yellow Bananas             0.79
   Granny Smith Apples        0.89
   Gala Apples                1.49
   Plums                      1.69
   Navel Oranges              1.29
   Fuji Apples                1.29

The collection contains the following elements after removing "Plums":
   KEY                        VALUE
   Seedless Watermelon        0.49
   Nectarine                  1.99
   Cranberries                5.98
   Plantain Bananas           1.49
   Honeydew Melon             0.59
   Pineapple                  1.49
   Strawberries               3.33
   Grapes                     1.99
   Braeburn Apples            1.49
   Peaches                    1.99
   Red Delicious Apples       0.99
   Golden Delicious Apples    1.29
   Yellow Bananas             0.79
```

```
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Navel Oranges             1.29
   Fuji Apples               1.29

The collection contains the following elements after it is cleared:
   KEY                       VALUE

*/
```

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.

A key cannot be `null`, but a value can.

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 HybridDictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the HybridDictionary, setting the Item[Object] property overwrites the old value. In contrast, the Add method does not modify existing elements.

When the number of elements becomes greater than the optimal size for a ListDictionary, the elements are copied from the ListDictionary to a Hashtable. However, this only happens once. If the collection is already stored in a Hashtable and the number of elements falls below the optimal size for a ListDictionary, the collection remains in the Hashtable.

This method is an O(1) operation.

See Also

Remove(Object)Remove(Object)
Item[Object]Item[Object]

HybridDictionary.Clear HybridDictionary.Clear

In this Article

Removes all entries from the HybridDictionary.

```
public void Clear ();
```

```
abstract member Clear : unit -> unit
override this.Clear : unit -> unit
```

Examples

The following code example adds to and removes elements from a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Displays the values in the HybridDictionary in three different ways.
      Console.WriteLine( "Initial contents of the HybridDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":"
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
```

```
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the HybridDictionary:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

The collection contains the following elements after removing "Plums":
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Navel Oranges             1.29
   Fuji Apples               1.29

The collection contains the following elements after it is cleared:
   KEY                       VALUE

*/
```

Remarks

Count is set to zero, and references to other objects from elements of the collection are also released.

If the collection is already stored in a Hashtable, the collection remains in the Hashtable.

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

HybridDictionary.Contains HybridDictionary.Contains

In this Article

Determines whether the HybridDictionary contains a specific key.

```
public bool Contains (object key);
```

```
abstract member Contains : obj -> bool
override this.Contains : obj -> bool
```

Parameters

key Object Object

The key to locate in the HybridDictionary.

Returns

Boolean Boolean

`true` if the HybridDictionary contains an entry with the specified key; otherwise, `false`.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example searches for an element in a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Displays the values in the HybridDictionary in three different ways.
      Console.WriteLine( "Initial contents of the HybridDictionary:" );
      PrintKeysAndValues( myCol );
```

```
      PrintKeysAndValues( myCol );

      // Searches for a key.
      if ( myCol.Contains( "Kiwis" ) )
         Console.WriteLine( "The collection contains the key \"Kiwis\"." );
      else
         Console.WriteLine( "The collection does not contain the key \"Kiwis\"." );
      Console.WriteLine();

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the HybridDictionary:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

The collection does not contain the key "Kiwis".

*/
```

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 `item` parameter on the objects in the collection.

See Also	IDictionaryIDictionary Performing Culture-Insensitive String Operations

HybridDictionary.CopyTo HybridDictionary.CopyTo

In this Article

Copies the HybridDictionary entries to a one-dimensional Array instance 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 HybridDictionary. 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 HybridDictionary is greater than the available space from `arrayIndex` to the end of the destination `array`.

InvalidCastException InvalidCastException

The type of the source HybridDictionary cannot be cast automatically to the type of the destination `array`.

Examples

The following code example copies the elements of a HybridDictionary to an array.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
```

```
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Displays the values in the HybridDictionary in three different ways.
      Console.WriteLine( "Initial contents of the HybridDictionary:" );
      PrintKeysAndValues( myCol );

      // Copies the HybridDictionary to an array with DictionaryEntry elements.
      DictionaryEntry[] myArr = new DictionaryEntry[myCol.Count];
      myCol.CopyTo( myArr, 0 );

      // Displays the values in the array.
      Console.WriteLine( "Displays the elements in the array:" );
      Console.WriteLine( "   KEY                       VALUE" );
      for ( int i = 0; i < myArr.Length; i++ )
         Console.WriteLine( "   {0,-25} {1}", myArr[i].Key, myArr[i].Value );
      Console.WriteLine();

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the HybridDictionary:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29
```

```
Displays the elements in the array:
   KEY                         VALUE
   Seedless Watermelon         0.49
   Nectarine                   1.99
   Cranberries                 5.98
   Plantain Bananas            1.49
   Honeydew Melon              0.59
   Pineapple                   1.49
   Strawberries                3.33
   Grapes                      1.99
   Braeburn Apples             1.49
   Peaches                     1.99
   Red Delicious Apples        0.99
   Golden Delicious Apples     1.29
   Yellow Bananas              0.79
   Granny Smith Apples         0.89
   Gala Apples                 1.49
   Plums                       1.69
   Navel Oranges               1.29
   Fuji Apples                 1.29

*/
```

Remarks

The elements are copied to the Array in the same order in which the enumerator iterates through the HybridDictionary.

To copy only the keys in the HybridDictionary, use `HybridDictionary.Keys.CopyTo`.

To copy only the values in the HybridDictionary, use `HybridDictionary.Values.CopyTo`.

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

See Also	ArrayArray GetEnumerator()GetEnumerator()

HybridDictionary.Count HybridDictionary.Count

In this Article

Gets the number of key/value pairs contained in the HybridDictionary.

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

```
member this.Count : int
```

Returns
Int32 Int32

The number of key/value pairs contained in the HybridDictionary.

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

Examples

The following code example enumerates the elements of a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );
```

```
   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( HybridDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
```

```
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Seedless Watermelon       0.49
   1     Nectarine                 1.99
   2     Cranberries               5.98
   3     Plantain Bananas          1.49
   4     Honeydew Melon            0.59
   5     Pineapple                 1.49
   6     Strawberries              3.33
   7     Grapes                    1.99
   8     Braeburn Apples           1.49
   9     Peaches                   1.99
   10    Red Delicious Apples      0.99
   11    Golden Delicious Apples   1.29
   12    Yellow Bananas            0.79
   13    Granny Smith Apples       0.89
   14    Gala Apples               1.49
   15    Plums                     1.69
   16    Navel Oranges             1.29
   17    Fuji Apples               1.29

*/
```

HybridDictionary.GetEnumerator HybridDictionary.GetEnumerator

In this Article

Returns an IDictionaryEnumerator that iterates through the HybridDictionary.

```
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 HybridDictionary.

Examples

The following code example enumerates the elements of a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );
```

```
   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( HybridDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
```

```
    Honeydew Melon            0.59
    Pineapple                 1.49
    Strawberries              3.33
    Grapes                    1.99
    Braeburn Apples           1.49
    Peaches                   1.99
    Red Delicious Apples      0.99
    Golden Delicious Apples   1.29
    Yellow Bananas            0.79
    Granny Smith Apples       0.89
    Gala Apples               1.49
    Plums                     1.69
    Navel Oranges             1.29
    Fuji Apples               1.29

 Displays the elements using the Keys, Values, Count, and Item properties:
    INDEX KEY                       VALUE
    0     Seedless Watermelon       0.49
    1     Nectarine                 1.99
    2     Cranberries               5.98
    3     Plantain Bananas          1.49
    4     Honeydew Melon            0.59
    5     Pineapple                 1.49
    6     Strawberries              3.33
    7     Grapes                    1.99
    8     Braeburn Apples           1.49
    9     Peaches                   1.99
    10    Red Delicious Apples      0.99
    11    Golden Delicious Apples   1.29
    12    Yellow Bananas            0.79
    13    Granny Smith Apples       0.89
    14    Gala Apples               1.49
    15    Plums                     1.69
    16    Navel Oranges             1.29
    17    Fuji Apples               1.29

 */
```

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 Also	IDictionaryEnumeratorIDictionaryEnumerator IEnumeratorIEnumerator

HybridDictionary HybridDictionary

In this Article

Overloads

HybridDictionary()	Creates an empty case-sensitive HybridDictionary.
HybridDictionary(Boolean) HybridDictionary(Boolean)	Creates an empty HybridDictionary with the specified case sensitivity.
HybridDictionary(Int32) HybridDictionary(Int32)	Creates a case-sensitive HybridDictionary with the specified initial size.
HybridDictionary(Int32, Boolean) HybridDictionary(Int32, Boolean)	Creates a HybridDictionary with the specified initial size and case sensitivity.

HybridDictionary()

Creates an empty case-sensitive HybridDictionary.

```
public HybridDictionary ();
```

Examples

The following code example demonstrates several of the properties and methods of HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
```

```
      myCol.Add( "Peaches", "1.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

      // Copies the HybridDictionary to an array with DictionaryEntry elements.
      DictionaryEntry[] myArr = new DictionaryEntry[myCol.Count];
      myCol.CopyTo( myArr, 0 );

      // Displays the values in the array.
      Console.WriteLine( "Displays the elements in the array:" );
      Console.WriteLine( "   KEY                       VALUE" );
      for ( int i = 0; i < myArr.Length; i++ )
         Console.WriteLine( "   {0,-25} {1}", myArr[i].Key, myArr[i].Value );
      Console.WriteLine();

      // Searches for a key.
      if ( myCol.Contains( "Kiwis" ) )
         Console.WriteLine( "The collection contains the key \"Kiwis\"." );
      else
         Console.WriteLine( "The collection does not contain the key \"Kiwis\"." );
      Console.WriteLine();

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":"
);
      PrintKeysAndValues1( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues1( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
```

```
   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( HybridDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Strawberries              3.33
   Yellow Bananas            0.79
   Cranberries               5.98
   Grapes                    1.99
   Granny Smith Apples       0.89
   Seedless Watermelon       0.49
   Honeydew Melon            0.59
   Red Delicious Apples      0.99
   Navel Oranges             1.29
   Fuji Apples               1.29
   Plantain Bananas          1.49
   Gala Apples               1.49
   Pineapple                 1.49
   Plums                     1.69
   Braeburn Apples           1.49
   Peaches                   1.99
   Golden Delicious Apples   1.29
   Nectarine                 1.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Strawberries              3.33
   Yellow Bananas            0.79
   Cranberries               5.98
   Grapes                    1.99
   Granny Smith Apples       0.89
   Seedless Watermelon       0.49
   Honeydew Melon            0.59
   Red Delicious Apples      0.99
   Navel Oranges             1.29
   Fuji Apples               1.29
   Plantain Bananas          1.49
   Gala Apples               1.49
   Pineapple                 1.49
   Plums                     1.69
   Braeburn Apples           1.49
   Peaches                   1.99
   Golden Delicious Apples   1.29
   Nectarine                 1.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Strawberries              3.33
   1     Yellow Bananas            0.79
   2     Cranberries               5.98
   3     Grapes                    1.99
   4     Granny Smith Apples       0.89
   5     Seedless Watermelon       0.49
```

```
    6     Honeydew Melon           0.59
    7     Red Delicious Apples     0.99
    8     Navel Oranges            1.29
    9     Fuji Apples              1.29
    10    Plantain Bananas         1.49
    11    Gala Apples              1.49
    12    Pineapple                1.49
    13    Plums                    1.69
    14    Braeburn Apples          1.49
    15    Peaches                  1.99
    16    Golden Delicious Apples  1.29
    17    Nectarine                1.99

Displays the elements in the array:
   KEY                        VALUE
   Strawberries               3.33
   Yellow Bananas             0.79
   Cranberries                5.98
   Grapes                     1.99
   Granny Smith Apples        0.89
   Seedless Watermelon        0.49
   Honeydew Melon             0.59
   Red Delicious Apples       0.99
   Navel Oranges              1.29
   Fuji Apples                1.29
   Plantain Bananas           1.49
   Gala Apples                1.49
   Pineapple                  1.49
   Plums                      1.69
   Braeburn Apples            1.49
   Peaches                    1.99
   Golden Delicious Apples    1.29
   Nectarine                  1.99

The collection does not contain the key "Kiwis".

The collection contains the following elements after removing "Plums":
   KEY                        VALUE
   Strawberries               3.33
   Yellow Bananas             0.79
   Cranberries                5.98
   Grapes                     1.99
   Granny Smith Apples        0.89
   Seedless Watermelon        0.49
   Honeydew Melon             0.59
   Red Delicious Apples       0.99
   Navel Oranges              1.29
   Fuji Apples                1.29
   Plantain Bananas           1.49
   Gala Apples                1.49
   Pineapple                  1.49
   Braeburn Apples            1.49
   Peaches                    1.99
   Golden Delicious Apples    1.29
   Nectarine                  1.99

The collection contains the following elements after it is cleared:
   KEY                        VALUE

*/
```

Remarks

By default, the collection is case-sensitive and uses the key's implementation of Object.GetHashCode as the hash code provider and the key's implementation of Object.Equals as the comparer.

The comparer determines whether two keys are equal. Every key in a HybridDictionary must be unique.

This constructor is an O(1) operation.

See Also | Performing Culture-Insensitive String Operations

HybridDictionary(Boolean) HybridDictionary(Boolean)

Creates an empty HybridDictionary with the specified case sensitivity.

```
public HybridDictionary (bool caseInsensitive);
```

```
new System.Collections.Specialized.HybridDictionary : bool ->
System.Collections.Specialized.HybridDictionary
```

Parameters

caseInsensitive | Boolean Boolean

A Boolean that denotes whether the HybridDictionary is case-insensitive.

Remarks

If `caseInsensitive` is `false`, the collection uses the key's implementations of Object.GetHashCode and Object.Equals. If `caseInsensitive` is `true`, the collection performs a simple ordinal case-insensitive comparison, which obeys the casing rules of the invariant culture only. For more information on the invariant culture, see System.Globalization.CultureInfo.

This constructor is an O(1) operation.

See Also | Performing Culture-Insensitive String Operations

HybridDictionary(Int32) HybridDictionary(Int32)

Creates a case-sensitive HybridDictionary with the specified initial size.

```
public HybridDictionary (int initialSize);
```

```
new System.Collections.Specialized.HybridDictionary : int ->
System.Collections.Specialized.HybridDictionary
```

Parameters

initialSize | Int32 Int32

The approximate number of entries that the HybridDictionary can initially contain.

Remarks

If the initial size of the collection is greater than the optimal size for a ListDictionary, the collection is stored in a Hashtable to avoid the overhead of copying elements from the ListDictionary to the Hashtable.

By default, the collection is case-sensitive and uses the key's implementation of Object.GetHashCode as the hash code provider and the key's implementation of Object.Equals as the comparer.

The comparer determines whether two keys are equal. Every key in a HybridDictionary must be unique.

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

See Also   Equals(Object)Equals(Object)
Performing Culture-Insensitive String Operations

HybridDictionary(Int32, Boolean) HybridDictionary(Int32, Boolean)

Creates a HybridDictionary with the specified initial size and case sensitivity.

```
public HybridDictionary (int initialSize, bool caseInsensitive);
```

```
new System.Collections.Specialized.HybridDictionary : int * bool ->
System.Collections.Specialized.HybridDictionary
```

Parameters

initialSize   Int32 Int32

The approximate number of entries that the HybridDictionary can initially contain.

caseInsensitive   Boolean Boolean

A Boolean that denotes whether the HybridDictionary is case-insensitive.

Remarks

If the initial size of the collection is greater than the optimal size for a ListDictionary, the collection is stored in a Hashtable to avoid the overhead of copying elements from the ListDictionary to the Hashtable.

If `caseInsensitive` is `false`, the collection uses the key's implementations of Object.GetHashCode and Object.Equals. If `caseInsensitive` is `true`, the collection performs a simple ordinal case-insensitive comparison, which obeys the casing rules of the invariant culture only. For more information on the invariant culture, see System.Globalization.CultureInfo.

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

See Also   Performing Culture-Insensitive String Operations

HybridDictionary.IEnumerable.GetEnumerator

In this Article

Returns an IEnumerator that iterates through the HybridDictionary.

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

Returns
IEnumerator

An IEnumerator for the HybridDictionary.

Examples

The following code example enumerates the elements of a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( HybridDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
```

```
    Peaches                    1.99
    Red Delicious Apples       0.99
    Golden Delicious Apples    1.29
    Yellow Bananas             0.79
    Granny Smith Apples        0.89
    Gala Apples                1.49
    Plums                      1.69
    Navel Oranges              1.29
    Fuji Apples                1.29

 Displays the elements using the Keys, Values, Count, and Item properties:
    INDEX KEY                        VALUE
    0     Seedless Watermelon        0.49
    1     Nectarine                  1.99
    2     Cranberries                5.98
    3     Plantain Bananas           1.49
    4     Honeydew Melon             0.59
    5     Pineapple                  1.49
    6     Strawberries               3.33
    7     Grapes                     1.99
    8     Braeburn Apples            1.49
    9     Peaches                    1.99
    10    Red Delicious Apples       0.99
    11    Golden Delicious Apples    1.29
    12    Yellow Bananas             0.79
    13    Granny Smith Apples        0.89
    14    Gala Apples                1.49
    15    Plums                      1.69
    16    Navel Oranges              1.29
    17    Fuji Apples                1.29

 */
```

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 Also	IDictionaryEnumerator IEnumerator

HybridDictionary.IsFixedSize HybridDictionary.IsFixedSize

In this Article

Gets a value indicating whether the HybridDictionary has a fixed size.

```
public bool IsFixedSize { get; }
```

```
member this.IsFixedSize : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Remarks

HybridDictionary implements the IsFixedSize property because it is required by the System.Collections.IDictionary interface.

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.

HybridDictionary.IsReadOnly HybridDictionary.IsReadOnly

In this Article

Gets a value indicating whether the HybridDictionary is read-only.

```
public bool IsReadOnly { get; }
```

```
member this.IsReadOnly : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Remarks

HybridDictionary implements the IsReadOnly property because it is required by the System.Collections.IDictionary 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.

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

HybridDictionary.IsSynchronized HybridDictionary.IsSynchronized

In this Article

Gets a value indicating whether the HybridDictionary is synchronized (thread safe).

```
public bool IsSynchronized { get; }
```

```
member this.IsSynchronized : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Examples

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration.

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

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

Remarks

HybridDictionary implements the IsSynchronized property because it is required by the System.Collections.ICollection interface.

Derived classes can provide a synchronized version of the HybridDictionary 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

SyncRootSyncRoot

HybridDictionary.Item[Object] HybridDictionary.Item[Object]

In this Article

Gets or sets the value associated 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 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 entry using the specified key.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example enumerates the elements of a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );
```

```
      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( HybridDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
```

```
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Seedless Watermelon       0.49
   1     Nectarine                 1.99
   2     Cranberries               5.98
   3     Plantain Bananas          1.49
   4     Honeydew Melon            0.59
   5     Pineapple                 1.49
   6     Strawberries              3.33
   7     Grapes                    1.99
   8     Braeburn Apples           1.49
   9     Peaches                   1.99
   10    Red Delicious Apples      0.99
   11    Golden Delicious Apples   1.29
   12    Yellow Bananas            0.79
   13    Granny Smith Apples       0.89
   14    Gala Apples               1.49
   15    Plums                     1.69
   16    Navel Oranges             1.29
   17    Fuji Apples               1.29

*/
```

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 HybridDictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the HybridDictionary, 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. 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 to determine if the key exists in the list.

The C# language uses the 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.

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

See Also	Add(Object, Object)Add(Object, Object) Performing Culture-Insensitive String Operations

HybridDictionary.Keys HybridDictionary.Keys

In this Article

Gets an ICollection containing the keys in the HybridDictionary.

```
public System.Collections.ICollection Keys { get; }
```

```
member this.Keys : System.Collections.ICollection
```

Returns

ICollection ICollection

An ICollection containing the keys in the HybridDictionary.

Examples

The following code example enumerates the elements of a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:" 
);
      PrintKeysAndValues3( myCol );

   }
```

```
   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( HybridDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
```

```
    Grapes                    1.99
    Braeburn Apples           1.49
    Peaches                   1.99
    Red Delicious Apples      0.99
    Golden Delicious Apples   1.29
    Yellow Bananas            0.79
    Granny Smith Apples       0.89
    Gala Apples               1.49
    Plums                     1.69
    Navel Oranges             1.29
    Fuji Apples               1.29

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Seedless Watermelon       0.49
   1     Nectarine                 1.99
   2     Cranberries               5.98
   3     Plantain Bananas          1.49
   4     Honeydew Melon            0.59
   5     Pineapple                 1.49
   6     Strawberries              3.33
   7     Grapes                    1.99
   8     Braeburn Apples           1.49
   9     Peaches                   1.99
   10    Red Delicious Apples      0.99
   11    Golden Delicious Apples   1.29
   12    Yellow Bananas            0.79
   13    Granny Smith Apples       0.89
   14    Gala Apples               1.49
   15    Plums                     1.69
   16    Navel Oranges             1.29
   17    Fuji Apples               1.29

*/
```

Remarks

The order of the values 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 HybridDictionary. Therefore, changes to the HybridDictionary continue to be reflected in the ICollection.

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

See Also	ICollectionICollection ValuesValues

HybridDictionary.Remove HybridDictionary.Remove

In this Article

Removes the entry with the specified key from the HybridDictionary.

```
public void Remove (object key);
```

```
abstract member Remove : obj -> unit
override this.Remove : obj -> unit
```

Parameters

key

Object Object

The key of the entry to remove.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example adds to and removes elements from a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Displays the values in the HybridDictionary in three different ways.
      Console.WriteLine( "Initial contents of the HybridDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":"
);
```

```
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the HybridDictionary:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

The collection contains the following elements after removing "Plums":
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Navel Oranges             1.29
   Fuji Apples               1.29

The collection contains the following elements after it is cleared:
   KEY                       VALUE

*/
```

```
*/
```

Remarks

If the HybridDictionary does not contain an element with the specified key, the HybridDictionary remains unchanged. No exception is thrown.

If the collection is already stored in a Hashtable and the number of elements falls below the optimal size for a ListDictionary, the collection remains in the Hashtable to avoid the overhead of copying elements from the Hashtable back to a ListDictionary.

This method is an O(1) operation.

See Also	Add(Object, Object)Add(Object, Object) Performing Culture-Insensitive String Operations

HybridDictionary.SyncRoot HybridDictionary.SyncRoot

In this Article

Gets an object that can be used to synchronize access to the HybridDictionary.

```
public object SyncRoot { get; }
```

```
member this.SyncRoot : obj
```

Returns

Object Object

An object that can be used to synchronize access to the HybridDictionary.

Examples

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration.

```
HybridDictionary myCollection = new HybridDictionary();
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 HybridDictionary using the SyncRoot property. The synchronizing code must perform operations on the SyncRoot of the HybridDictionary, not directly on the HybridDictionary. 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 HybridDictionary 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

IsSynchronizedIsSynchronized

HybridDictionary.Values HybridDictionary.Values

In this Article

Gets an ICollection containing the values in the HybridDictionary.

```
public System.Collections.ICollection Values { get; }
```

```
member this.Values : System.Collections.ICollection
```

Returns

ICollection ICollection

An ICollection containing the values in the HybridDictionary.

Examples

The following code example enumerates the elements of a HybridDictionary.

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

public class SamplesHybridDictionary  {

   public static void Main()  {

      // Creates and initializes a new HybridDictionary.
      HybridDictionary myCol = new HybridDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );
      myCol.Add( "Plantain Bananas", "1.49" );
      myCol.Add( "Yellow Bananas", "0.79" );
      myCol.Add( "Strawberries", "3.33" );
      myCol.Add( "Cranberries", "5.98" );
      myCol.Add( "Navel Oranges", "1.29" );
      myCol.Add( "Grapes", "1.99" );
      myCol.Add( "Honeydew Melon", "0.59" );
      myCol.Add( "Seedless Watermelon", "0.49" );
      myCol.Add( "Pineapple", "1.49" );
      myCol.Add( "Nectarine", "1.99" );
      myCol.Add( "Plums", "1.69" );
      myCol.Add( "Peaches", "1.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }
```

```
   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( HybridDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
   Braeburn Apples           1.49
   Peaches                   1.99
   Red Delicious Apples      0.99
   Golden Delicious Apples   1.29
   Yellow Bananas            0.79
   Granny Smith Apples       0.89
   Gala Apples               1.49
   Plums                     1.69
   Navel Oranges             1.29
   Fuji Apples               1.29

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Seedless Watermelon       0.49
   Nectarine                 1.99
   Cranberries               5.98
   Plantain Bananas          1.49
   Honeydew Melon            0.59
   Pineapple                 1.49
   Strawberries              3.33
   Grapes                    1.99
```

```
    Grapes                    1.99
    Braeburn Apples           1.49
    Peaches                   1.99
    Red Delicious Apples      0.99
    Golden Delicious Apples   1.29
    Yellow Bananas            0.79
    Granny Smith Apples       0.89
    Gala Apples               1.49
    Plums                     1.69
    Navel Oranges             1.29
    Fuji Apples               1.29

Displays the elements using the Keys, Values, Count, and Item properties:
    INDEX KEY                       VALUE
    0     Seedless Watermelon       0.49
    1     Nectarine                 1.99
    2     Cranberries               5.98
    3     Plantain Bananas          1.49
    4     Honeydew Melon            0.59
    5     Pineapple                 1.49
    6     Strawberries              3.33
    7     Grapes                    1.99
    8     Braeburn Apples           1.49
    9     Peaches                   1.99
    10    Red Delicious Apples      0.99
    11    Golden Delicious Apples   1.29
    12    Yellow Bananas            0.79
    13    Granny Smith Apples       0.89
    14    Gala Apples               1.49
    15    Plums                     1.69
    16    Navel Oranges             1.29
    17    Fuji Apples               1.29

*/
```

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 HybridDictionary. Therefore, changes to the HybridDictionary continue to be reflected in the ICollection.

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

See Also	ICollectionICollection KeysKeys

INotifyCollectionChanged INotifyCollectionChanged Interface

Notifies listeners of dynamic changes, such as when an item is added and removed or the whole list is cleared.

Declaration

```
public interface INotifyCollectionChanged
```

```
type INotifyCollectionChanged = interface
```

Inheritance Hierarchy

None

Remarks

You can enumerate over any collection that implements the IEnumerable interface. However, to set up dynamic bindings so that insertions or deletions in the collection update the UI automatically, the collection must implement the INotifyCollectionChanged interface. This interface exposes the CollectionChanged event that must be raised whenever the underlying collection changes.

WPF provides the ObservableCollection<T> class, which is a built-in implementation of a data collection that exposes the INotifyCollectionChanged interface. For an example, see How to: Create and Bind to an ObservableCollection.

The individual data objects within the collection must satisfy the requirements described in the Binding Sources Overview.

Before implementing your own collection, consider using ObservableCollection<T> or one of the existing collection classes, such as List<T>, Collection<T>, and BindingList<T>, among many others.

If you have an advanced scenario and want to implement your own collection, consider using IList, which provides a non-generic collection of objects that can be individually accessed by index and provides the best performance.

Events

`CollectionChanged`

`CollectionChanged`

Occurs when the collection changes.

See Also

INotifyCollectionChanged.CollectionChanged INotifyCollectionChanged.CollectionChanged

In this Article

Occurs when the collection changes.

```
event System.Collections.Specialized.NotifyCollectionChangedEventHandler CollectionChanged;
```

```
member this.CollectionChanged : System.Collections.Specialized.NotifyCollectionChangedEventHandler
```

Remarks

The event handler receives an argument of type NotifyCollectionChangedEventArgs,which contains data that is related to this event.

IOrderedDictionary IOrderedDictionary Interface

Represents an indexed collection of key/value pairs.

Declaration

```
public interface IOrderedDictionary : System.Collections.IDictionary
```

```
type IOrderedDictionary = interface
    interface IDictionary
    interface ICollection
    interface IEnumerable
```

Inheritance Hierarchy

`None`

Remarks

IOrderedDictionary elements can be accessed either with the key or with the index.

Each element is a key/value pair stored in a DictionaryEntry structure.

Each pair must have a unique key that is not `null`, but the value can be `null` and does not have to be unique. The IOrderedDictionary interface allows the contained keys and values to be enumerated, but it does not imply any particular sort order.

The `foreach` statement of the C# language (`For Each` in Visual Basic) returns an object of the type of the elements in the collection. Because 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, as the following example shows.

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

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

Properties

`Item[Int32]`

`Item[Int32]`

Gets or sets the element at the specified index.

Methods

`GetEnumerator()`

`GetEnumerator()`

Returns an enumerator that iterates through the IOrderedDictionary collection.

`Insert(Int32, Object, Object)`

`Insert(Int32, Object, Object)`

Inserts a key/value pair into the collection at the specified index.

`RemoveAt(Int32)`

`RemoveAt(Int32)`

Removes the element at the specified index.

See Also

ICollection ICollection
IDictionary IDictionary

IOrderedDictionary.GetEnumerator IOrderedDictionary.GetEnumerator

In this Article

Returns an enumerator that iterates through the IOrderedDictionary collection.

```
public System.Collections.IDictionaryEnumerator GetEnumerator ();
```

```
abstract member GetEnumerator : unit -> System.Collections.IDictionaryEnumerator
```

Returns
IDictionaryEnumerator IDictionaryEnumerator

An IDictionaryEnumerator for the entire IOrderedDictionary collection.

Examples

The following code example demonstrates the implementation of a simple IOrderedDictionary based on the ArrayList class. The implemented IOrderedDictionary stores first names as the keys and last names as the values, with the added requirement that each first name is unique. This code is part of a larger code example provided for the IOrderedDictionary class.

```
public class People : IOrderedDictionary
{
    private ArrayList _people;

    public People(int numItems)
    {
        _people = new ArrayList(numItems);
    }

    public int IndexOfKey(object key)
    {
        for (int i = 0; i < _people.Count; i++)
        {
            if (((DictionaryEntry)_people[i]).Key == key)
                return i;
        }

        // key not found, reutrn -1.
        return -1;
    }

    public object this[object key]
    {
        get
        {
            return ((DictionaryEntry)_people[IndexOfKey(key)]).Value;
        }
        set
        {
            _people[IndexOfKey(key)] = new DictionaryEntry(key, value);
        }
    }

    // IOrderedDictionary Members
    public IDictionaryEnumerator GetEnumerator()
    {
        return new PeopleEnum(_people);
    }
```

```
    public void Insert(int index, object key, object value)
    {
        if (IndexOfKey(key) != -1)
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Insert(index, new DictionaryEntry(key, value));
    }

    public void RemoveAt(int index)
    {
        _people.RemoveAt(index);
    }

    public object this[int index]
    {
        get
        {
            return ((DictionaryEntry)_people[index]).Value;
        }
        set
        {
            object key = ((DictionaryEntry)_people[index]).Key;
            _people[index] = new DictionaryEntry(Keys, value);
        }
    }
    // IDictionary Members

    public void Add(object key, object value)
    {
        if (IndexOfKey(key) != -1)
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Add(new DictionaryEntry(key, value));
    }

    public void Clear()
    {
        _people.Clear();
    }

    public bool Contains(object key)
    {
        if (IndexOfKey(key) == -1)
        {
            return false;
        }
        else
        {
            return true;
        }
    }

    public bool IsFixedSize
    {
        get
        {
            return false;
        }
    }

    public bool IsReadOnly
```

```
{
    get
    {
        return false;
    }
}

public ICollection Keys
{
    get
    {
        ArrayList KeyCollection = new ArrayList(_people.Count);
        for (int i = 0; i < _people.Count; i++)
        {
            KeyCollection.Add( ((DictionaryEntry)_people[i]).Key );
        }
        return KeyCollection;
    }
}

public void Remove(object key)
{
    _people.RemoveAt(IndexOfKey(key));
}

public ICollection Values
{
    get
    {
        ArrayList ValueCollection = new ArrayList(_people.Count);
        for (int i = 0; i < _people.Count; i++)
        {
            ValueCollection.Add( ((DictionaryEntry)_people[i]).Value );
        }
        return ValueCollection;
    }
}


// ICollection Members

public void CopyTo(Array array, int index)
{
    _people.CopyTo(array, index);
}

public int Count
{
    get
    {
        return _people.Count;
    }
}

public bool IsSynchronized
{
    get
    {
        return _people.IsSynchronized;
    }
}

public object SyncRoot
{
    get
```

```
        {
            return _people.SyncRoot;
        }
    }

    // IEnumerable Members

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new PeopleEnum(_people);
    }
}

public class PeopleEnum : IDictionaryEnumerator
{
    public ArrayList _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(ArrayList list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Count);
    }

    public void Reset()
    {
        position = -1;
    }

    public object Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }

    public DictionaryEntry Entry
    {
        get
        {
            return (DictionaryEntry)Current;
        }
    }

    public object Key
    {
        get
        {
            try
            {
```

```
            {
                return ((DictionaryEntry)_people[position]).Key;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }

    public object Value
    {
        get
        {
            try
            {
                return ((DictionaryEntry)_people[position]).Value;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}
```

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.

IOrderedDictionary.Insert IOrderedDictionary.Insert

In this Article

Inserts a key/value pair into the collection at the specified index.

```
public void Insert (int index, object key, object value);
```

```
abstract member Insert : int * obj * obj -> unit
```

Parameters

index Int32 Int32

The zero-based index at which the key/value pair should be inserted.

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. The value can be `null`.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than 0.

-or-

`index` is greater than Count.

ArgumentNullException ArgumentNullException

`key` is `null`.

ArgumentException ArgumentException

An element with the same key already exists in the IOrderedDictionary collection.

NotSupportedException NotSupportedException

The IOrderedDictionary collection is read-only.

-or-

The IOrderedDictionary collection has a fixed size.

Examples

The following code example demonstrates the implementation of a simple IOrderedDictionary based on the ArrayList class. The implemented IOrderedDictionary stores first names as the keys and last names as the values, with the added requirement that each first name is unique. This code is part of a larger code example provided for the IOrderedDictionary class.

```
public class People : IOrderedDictionary
{
    private ArrayList _people;

    public People(int numItems)
```

```
    {
        _people = new ArrayList(numItems);
    }

    public int IndexOfKey(object key)
    {
        for (int i = 0; i < _people.Count; i++)
        {
            if (((DictionaryEntry)_people[i]).Key == key)
                return i;
        }

        // key not found, reutrn -1.
        return -1;
    }

    public object this[object key]
    {
        get
        {
            return ((DictionaryEntry)_people[IndexOfKey(key)]).Value;
        }
        set
        {
            _people[IndexOfKey(key)] = new DictionaryEntry(key, value);
        }
    }

    // IOrderedDictionary Members
    public IDictionaryEnumerator GetEnumerator()
    {
        return new PeopleEnum(_people);
    }

    public void Insert(int index, object key, object value)
    {
        if (IndexOfKey(key) != -1)
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Insert(index, new DictionaryEntry(key, value));
    }

    public void RemoveAt(int index)
    {
        _people.RemoveAt(index);
    }

    public object this[int index]
    {
        get
        {
            return ((DictionaryEntry)_people[index]).Value;
        }
        set
        {
            object key = ((DictionaryEntry)_people[index]).Key;
            _people[index] = new DictionaryEntry(Keys, value);
        }
    }
    // IDictionary Members

    public void Add(object key, object value)
    {
        if (IndexOfKey(key) != -1)
```

```
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Add(new DictionaryEntry(key, value));
    }

    public void Clear()
    {
        _people.Clear();
    }

    public bool Contains(object key)
    {
        if (IndexOfKey(key) == -1)
        {
            return false;
        }
        else
        {
            return true;
        }
    }

    public bool IsFixedSize
    {
        get
        {
            return false;
        }
    }

    public bool IsReadOnly
    {
        get
        {
            return false;
        }
    }

    public ICollection Keys
    {
        get
        {
            ArrayList KeyCollection = new ArrayList(_people.Count);
            for (int i = 0; i < _people.Count; i++)
            {
                KeyCollection.Add( ((DictionaryEntry)_people[i]).Key );
            }
            return KeyCollection;
        }
    }

    public void Remove(object key)
    {
        _people.RemoveAt(IndexOfKey(key));
    }

    public ICollection Values
    {
        get
        {
            ArrayList ValueCollection = new ArrayList(_people.Count);
            for (int i = 0; i < _people.Count; i++)
            {
```

```
                ValueCollection.Add( ((DictionaryEntry)_people[i]).Value );
            }
            return ValueCollection;
        }
    }


    // ICollection Members

    public void CopyTo(Array array, int index)
    {
        _people.CopyTo(array, index);
    }

    public int Count
    {
        get
        {
            return _people.Count;
        }
    }

    public bool IsSynchronized
    {
        get
        {
            return _people.IsSynchronized;
        }
    }

    public object SyncRoot
    {
        get
        {
            return _people.SyncRoot;
        }
    }

    // IEnumerable Members

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new PeopleEnum(_people);
    }
}

public class PeopleEnum : IDictionaryEnumerator
{
    public ArrayList _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(ArrayList list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Count);
    }

    public void Reset()
```

```
    public void Reset()
    {
        position = -1;
    }

    public object Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }

    public DictionaryEntry Entry
    {
        get
        {
            return (DictionaryEntry)Current;
        }
    }

    public object Key
    {
        get
        {
            try
            {
                return ((DictionaryEntry)_people[position]).Key;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }

    public object Value
    {
        get
        {
            try
            {
                return ((DictionaryEntry)_people[position]).Value;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}
```

Remarks

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

If the `index` parameter is equal to Count, the `value` parameter is added to the end of the IOrderedDictionary collection.

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 together, such as a hash table.

IOrderedDictionary.Item[Int32] IOrderedDictionary.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 less than 0.

-or-

`index` is equal to or greater than Count.

Examples

The following code example demonstrates the implementation of a simple IOrderedDictionary based on the ArrayList class. The implemented IOrderedDictionary stores first names as the keys and last names as the values, with the added requirement that each first name is unique. This code is part of a larger code example provided for the IOrderedDictionary class.

```
public class People : IOrderedDictionary
{
    private ArrayList _people;

    public People(int numItems)
    {
        _people = new ArrayList(numItems);
    }

    public int IndexOfKey(object key)
    {
        for (int i = 0; i < _people.Count; i++)
        {
            if (((DictionaryEntry)_people[i]).Key == key)
                return i;
        }

        // key not found, reutrn -1.
        return -1;
    }

    public object this[object key]
    {
```

```
    {
        get
        {
            return ((DictionaryEntry)_people[IndexOfKey(key)]).Value;
        }
        set
        {
            _people[IndexOfKey(key)] = new DictionaryEntry(key, value);
        }
    }

    // IOrderedDictionary Members
    public IDictionaryEnumerator GetEnumerator()
    {
        return new PeopleEnum(_people);
    }

    public void Insert(int index, object key, object value)
    {
        if (IndexOfKey(key) != -1)
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Insert(index, new DictionaryEntry(key, value));
    }

    public void RemoveAt(int index)
    {
        _people.RemoveAt(index);
    }

    public object this[int index]
    {
        get
        {
            return ((DictionaryEntry)_people[index]).Value;
        }
        set
        {
            object key = ((DictionaryEntry)_people[index]).Key;
            _people[index] = new DictionaryEntry(Keys, value);
        }
    }
    // IDictionary Members

    public void Add(object key, object value)
    {
        if (IndexOfKey(key) != -1)
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Add(new DictionaryEntry(key, value));
    }

    public void Clear()
    {
        _people.Clear();
    }

    public bool Contains(object key)
    {
        if (IndexOfKey(key) == -1)
        {
            return false;
```

```
        }
        else
        {
            return true;
        }
    }

    public bool IsFixedSize
    {
        get
        {
            return false;
        }
    }

    public bool IsReadOnly
    {
        get
        {
            return false;
        }
    }

    public ICollection Keys
    {
        get
        {
            ArrayList KeyCollection = new ArrayList(_people.Count);
            for (int i = 0; i < _people.Count; i++)
            {
                KeyCollection.Add( ((DictionaryEntry)_people[i]).Key );
            }
            return KeyCollection;
        }
    }

    public void Remove(object key)
    {
        _people.RemoveAt(IndexOfKey(key));
    }

    public ICollection Values
    {
        get
        {
            ArrayList ValueCollection = new ArrayList(_people.Count);
            for (int i = 0; i < _people.Count; i++)
            {
                ValueCollection.Add( ((DictionaryEntry)_people[i]).Value );
            }
            return ValueCollection;
        }
    }


    // ICollection Members

    public void CopyTo(Array array, int index)
    {
        _people.CopyTo(array, index);
    }

    public int Count
    {
        get
        {
```

```
        {
            return _people.Count;
        }
    }

    public bool IsSynchronized
    {
        get
        {
            return _people.IsSynchronized;
        }
    }

    public object SyncRoot
    {
        get
        {
            return _people.SyncRoot;
        }
    }

    // IEnumerable Members

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new PeopleEnum(_people);
    }
}

public class PeopleEnum : IDictionaryEnumerator
{
    public ArrayList _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(ArrayList list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Count);
    }

    public void Reset()
    {
        position = -1;
    }

    public object Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
```

```
    }

    public DictionaryEntry Entry
    {
        get
        {
            return (DictionaryEntry)Current;
        }
    }

    public object Key
    {
        get
        {
            try
            {
                return ((DictionaryEntry)_people[position]).Key;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }

    public object Value
    {
        get
        {
            try
            {
                return ((DictionaryEntry)_people[position]).Value;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}
```

Remarks

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

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.

This property allows you to access a specific element in the collection by using the following syntax:

```
obj = myOrderedDictionary[index];
```

IOrderedDictionary.RemoveAt IOrderedDictionary.RemoveAt

In this Article

Removes the element at the specified index.

```
public void RemoveAt (int index);
```

```
abstract member RemoveAt : int -> unit
```

Parameters

index Int32 Int32

The zero-based index of the element to remove.

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than 0.

-or-

`index` is equal to or greater than Count.

NotSupportedException NotSupportedException

The IOrderedDictionary collection is read-only.

-or-

The IOrderedDictionary collection has a fixed size.

Examples

The following code example demonstrates the implementation of a simple IOrderedDictionary based on the ArrayList class. The implemented IOrderedDictionary stores first names as the keys and last names as the values, with the added requirement that each first name is unique. This code is part of a larger code example provided for the IOrderedDictionary class.

```
public class People : IOrderedDictionary
{
    private ArrayList _people;

    public People(int numItems)
    {
        _people = new ArrayList(numItems);
    }

    public int IndexOfKey(object key)
    {
        for (int i = 0; i < _people.Count; i++)
        {
            if (((DictionaryEntry)_people[i]).Key == key)
                return i;
        }

        // key not found, reutrn -1.
        return -1;
    }
```

```
    }

    public object this[object key]
    {
        get
        {
            return ((DictionaryEntry)_people[IndexOfKey(key)]).Value;
        }
        set
        {
            _people[IndexOfKey(key)] = new DictionaryEntry(key, value);
        }
    }

    // IOrderedDictionary Members
    public IDictionaryEnumerator GetEnumerator()
    {
        return new PeopleEnum(_people);
    }

    public void Insert(int index, object key, object value)
    {
        if (IndexOfKey(key) != -1)
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Insert(index, new DictionaryEntry(key, value));
    }

    public void RemoveAt(int index)
    {
        _people.RemoveAt(index);
    }

    public object this[int index]
    {
        get
        {
            return ((DictionaryEntry)_people[index]).Value;
        }
        set
        {
            object key = ((DictionaryEntry)_people[index]).Key;
            _people[index] = new DictionaryEntry(Keys, value);
        }
    }
    // IDictionary Members

    public void Add(object key, object value)
    {
        if (IndexOfKey(key) != -1)
        {
            throw new ArgumentException("An element with the same key already exists in the
collection.");
        }
        _people.Add(new DictionaryEntry(key, value));
    }

    public void Clear()
    {
        _people.Clear();
    }

    public bool Contains(object key)
    {
```

```
        if (IndexOfKey(key) == -1)
        {
            return false;
        }
        else
        {
            return true;
        }
    }

    public bool IsFixedSize
    {
        get
        {
            return false;
        }
    }

    public bool IsReadOnly
    {
        get
        {
            return false;
        }
    }

    public ICollection Keys
    {
        get
        {
            ArrayList KeyCollection = new ArrayList(_people.Count);
            for (int i = 0; i < _people.Count; i++)
            {
                KeyCollection.Add( ((DictionaryEntry)_people[i]).Key );
            }
            return KeyCollection;
        }
    }

    public void Remove(object key)
    {
        _people.RemoveAt(IndexOfKey(key));
    }

    public ICollection Values
    {
        get
        {
            ArrayList ValueCollection = new ArrayList(_people.Count);
            for (int i = 0; i < _people.Count; i++)
            {
                ValueCollection.Add( ((DictionaryEntry)_people[i]).Value );
            }
            return ValueCollection;
        }
    }


    // ICollection Members

    public void CopyTo(Array array, int index)
    {
        _people.CopyTo(array, index);
    }

    public int Count
```

```
    public int Count
    {
        get
        {
            return _people.Count;
        }
    }

    public bool IsSynchronized
    {
        get
        {
            return _people.IsSynchronized;
        }
    }

    public object SyncRoot
    {
        get
        {
            return _people.SyncRoot;
        }
    }

    // IEnumerable Members

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new PeopleEnum(_people);
    }
}

public class PeopleEnum : IDictionaryEnumerator
{
    public ArrayList _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(ArrayList list)
    {
        _people = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _people.Count);
    }

    public void Reset()
    {
        position = -1;
    }

    public object Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
```

```
                throw new InvalidOperationException();
            }
        }
    }

    public DictionaryEntry Entry
    {
        get
        {
            return (DictionaryEntry)Current;
        }
    }

    public object Key
    {
        get
        {
            try
            {
                return ((DictionaryEntry)_people[position]).Key;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }

    public object Value
    {
        get
        {
            try
            {
                return ((DictionaryEntry)_people[position]).Value;
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}
```

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 together, such as a hash table.

ListDictionary ListDictionary Class

Implements `IDictionary` using a singly linked list. Recommended for collections that typically include fewer than 10 items.

Declaration

```
[Serializable]
public class ListDictionary : System.Collections.IDictionary
```

```
type ListDictionary = class
    interface IDictionary
    interface ICollection
    interface IEnumerable
```

Inheritance Hierarchy

`Object` `Object`

Remarks

This is a simple implementation of IDictionary using a singly linked list. It is smaller and faster than a Hashtable if the number of elements is 10 or less. This should not be used if performance is important for large numbers of elements.

Items in a ListDictionary are not in any guaranteed order; code should not depend on the current order. The ListDictionary is implemented for fast keyed retrieval; the actual internal order of items is implementation-dependent and could change in future versions of the product.

Members, such as Item[Object], Add, Remove, and Contains are O(`n`) operations, where `n` is Count.

A key cannot be `null`, but a value can.

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 ListDictionary 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 myListDictionary)
{
    //...
}
```

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

Constructors

`ListDictionary()`

`ListDictionary()`

Creates an empty ListDictionary using the default comparer.

`ListDictionary(IComparer)`

`ListDictionary(IComparer)`

Creates an empty ListDictionary using the specified comparer.

Properties

`Count`

`Count`

Gets the number of key/value pairs contained in the ListDictionary.

`IsFixedSize`

`IsFixedSize`

Gets a value indicating whether the ListDictionary has a fixed size.

`IsReadOnly`

`IsReadOnly`

Gets a value indicating whether the ListDictionary is read-only.

`IsSynchronized`

`IsSynchronized`

Gets a value indicating whether the ListDictionary 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 ListDictionary.

`SyncRoot`

`SyncRoot`

Gets an object that can be used to synchronize access to the ListDictionary.

`Values`

`Values`

Gets an ICollection containing the values in the ListDictionary.

Methods

`Add(Object, Object)`

`Add(Object, Object)`

Adds an entry with the specified key and value into the ListDictionary.

`Clear()`

`Clear()`

Removes all entries from the ListDictionary.

`Contains(Object)`

`Contains(Object)`

Determines whether the ListDictionary contains a specific key.

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

Copies the ListDictionary entries to a one-dimensional Array instance at the specified index.

`GetEnumerator()`

`GetEnumerator()`

Returns an IDictionaryEnumerator that iterates through the ListDictionary.

`Remove(Object)`

`Remove(Object)`

Removes the entry with the specified key from the ListDictionary.

`IEnumerable.GetEnumerator()`

`IEnumerable.GetEnumerator()`

Returns an IEnumerator that iterates through the ListDictionary.

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 ListDictionary, but derived classes can create their own synchronized versions of the ListDictionary 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

IDictionary IDictionary
IDictionary IDictionary

ListDictionary.Add ListDictionary.Add

In this Article

Adds an entry with the specified key and value into the ListDictionary.

```
public 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 entry to add.

value

Object Object

The value of the entry to add. The value can be `null`.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

ArgumentException ArgumentException

An entry with the same key already exists in the ListDictionary.

Examples

The following code example adds to and removes elements from a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Displays the values in the ListDictionary in three different ways.
      Console.WriteLine( "Initial contents of the ListDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":"
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
```

```
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}


/*
This code produces the following output.

Initial contents of the ListDictionary:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

The collection contains the following elements after removing "Plums":
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

The collection contains the following elements after it is cleared:
   KEY                       VALUE

*/
```

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 ListDictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the ListDictionary, setting the Item[Object] property overwrites the old value. In contrast, the Add method does not modify existing elements.

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

See Also

Remove(Object)Remove(Object)
Item[Object]Item[Object]

ListDictionary.Clear ListDictionary.Clear

In this Article

Removes all entries from the ListDictionary.

```
public void Clear ();
```

```
abstract member Clear : unit -> unit
override this.Clear : unit -> unit
```

Examples

The following code example adds to and removes elements from a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Displays the values in the ListDictionary in three different ways.
      Console.WriteLine( "Initial contents of the ListDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":"
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}


/*
This code produces the following output.

Initial contents of the ListDictionary:
   KEY                       VALUE
```

```
   KEY                        VALUE
   Braeburn Apples            1.49
   Fuji Apples                1.29
   Gala Apples                1.49
   Golden Delicious Apples    1.29
   Granny Smith Apples        0.89
   Red Delicious Apples       0.99

The collection contains the following elements after removing "Plums":
   KEY                        VALUE
   Braeburn Apples            1.49
   Fuji Apples                1.29
   Gala Apples                1.49
   Golden Delicious Apples    1.29
   Granny Smith Apples        0.89
   Red Delicious Apples       0.99

The collection contains the following elements after it is cleared:
   KEY                        VALUE

*/
```

Remarks

Count is set to zero, and references to other objects from elements of the collection are also released.

This method is an O(1) operation.

ListDictionary.Contains ListDictionary.Contains

In this Article

Determines whether the ListDictionary contains a specific key.

```
public bool Contains (object key);
```

```
abstract member Contains : obj -> bool
override this.Contains : obj -> bool
```

Parameters

key Object Object

The key to locate in the ListDictionary.

Returns

Boolean Boolean

`true` if the ListDictionary contains an entry with the specified key; otherwise, `false`.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example searches for an element in a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Displays the values in the ListDictionary in three different ways.
      Console.WriteLine( "Initial contents of the ListDictionary:" );
      PrintKeysAndValues( myCol );

      // Searches for a key.
      if ( myCol.Contains( "Kiwis" ) )
         Console.WriteLine( "The collection contains the key \"Kiwis\"." );
      else
         Console.WriteLine( "The collection does not contain the key \"Kiwis\"." );
      Console.WriteLine();

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}


/*
This code produces the following output.

Initial contents of the ListDictionary:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

The collection does not contain the key "Kiwis".

*/
```

Remarks

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 `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 `item` parameter on the objects in the collection.

See Also	IDictionaryIDictionary Performing Culture-Insensitive String Operations

ListDictionary.CopyTo ListDictionary.CopyTo

In this Article

Copies the ListDictionary entries to a one-dimensional Array instance 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 ListDictionary. 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 ListDictionary is greater than the available space from `index` to the end of the destination `array`.

InvalidCastException InvalidCastException

The type of the source ListDictionary cannot be cast automatically to the type of the destination `array`.

Examples

The following code example copies the elements of a ListDictionary to an array.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
```

```
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Displays the values in the ListDictionary in three different ways.
      Console.WriteLine( "Initial contents of the ListDictionary:" );
      PrintKeysAndValues( myCol );

      // Copies the ListDictionary to an array with DictionaryEntry elements.
      DictionaryEntry[] myArr = new DictionaryEntry[myCol.Count];
      myCol.CopyTo( myArr, 0 );

      // Displays the values in the array.
      Console.WriteLine( "Displays the elements in the array:" );
      Console.WriteLine( "   KEY                       VALUE" );
      for ( int i = 0; i < myArr.Length; i++ )
         Console.WriteLine( "   {0,-25} {1}", myArr[i].Key, myArr[i].Value );
      Console.WriteLine();

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}


/*
This code produces the following output.

Initial contents of the ListDictionary:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements in the array:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

*/
```

Remarks

The elements are copied to the Array in the same order in which the enumerator iterates through the ListDictionary.

To copy only the keys in the ListDictionary, use `ListDictionary.Keys.CopyTo`.

To copy only the values in the ListDictionary, use `ListDictionary.Values.CopyTo`.

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

See Also	ArrayArray GetEnumerator()GetEnumerator()

ListDictionary.Count ListDictionary.Count

In this Article

Gets the number of key/value pairs contained in the ListDictionary.

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

```
member this.Count : int
```

Returns
Int32 Int32

The number of key/value pairs contained in the ListDictionary.

Examples

The following code example enumerates the elements of a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
```

```
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( ListDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Braeburn Apples           1.49
   1     Fuji Apples               1.29
   2     Gala Apples               1.49
   3     Golden Delicious Apples   1.29
   4     Granny Smith Apples       0.89
   5     Red Delicious Apples      0.99

*/
```

Remarks

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

ListDictionary.GetEnumerator ListDictionary.GetEnumerator

In this Article

Returns an IDictionaryEnumerator that iterates through the ListDictionary.

```
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 ListDictionary.

Examples

The following code example enumerates the elements of a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( ListDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Braeburn Apples           1.49
   1     Fuji Apples               1.29
   2     Gala Apples               1.49
   3     Golden Delicious Apples   1.29
   4     Granny Smith Apples       0.89
   5     Red Delicious Apples      0.99

*/
```

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 Also	IDictionaryEnumeratorIDictionaryEnumerator IEnumeratorIEnumerator

ListDictionary.IEnumerable.GetEnumerator

In this Article

Returns an IEnumerator that iterates through the ListDictionary.

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

Returns
IEnumerator

An IEnumerator for the ListDictionary.

Examples

The following code example enumerates the elements of a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
```

```
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( ListDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Braeburn Apples           1.49
   1     Fuji Apples               1.29
   2     Gala Apples               1.49
   3     Golden Delicious Apples   1.29
   4     Granny Smith Apples       0.89
   5     Red Delicious Apples      0.99

*/
```

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 Also	IDictionaryEnumerator IEnumerator

ListDictionary.IsFixedSize ListDictionary.IsFixedSize

In this Article

Gets a value indicating whether the ListDictionary has a fixed size.

```
public bool IsFixedSize { get; }
```

```
member this.IsFixedSize : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Remarks

ListDictionary implements the IsFixedSize property because it is required by the System.Collections.IDictionary interface.

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.

ListDictionary.IsReadOnly ListDictionary.IsReadOnly

In this Article

Gets a value indicating whether the ListDictionary is read-only.

```
public bool IsReadOnly { get; }
```

```
member this.IsReadOnly : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Remarks

ListDictionary implements the IsReadOnly property because it is required by the System.Collections.IDictionary 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.

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

ListDictionary.IsSynchronized ListDictionary.IsSynchronized

In this Article

Gets a value indicating whether the ListDictionary is synchronized (thread safe).

```
public bool IsSynchronized { get; }
```

```
member this.IsSynchronized : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Examples

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration.

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

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

Remarks

ListDictionary implements the IsSynchronized property because it is required by the System.Collections.ICollection interface.

Derived classes can provide a synchronized version of the ListDictionary 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

SyncRootSyncRoot

ListDictionary.Item[Object] ListDictionary.Item[Object]

In this Article

Gets or sets the value associated 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 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 entry using the specified key.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example enumerates the elements of a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }
```

```
   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( ListDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Braeburn Apples           1.49
   1     Fuji Apples               1.29
   2     Gala Apples               1.49
   3     Golden Delicious Apples   1.29
   4     Granny Smith Apples       0.89
   5     Red Delicious Apples      0.99

*/
```

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 ListDictionary; for example, `myCollection["myNonexistentKey"] = myValue`. However, if the specified key already exists in the ListDictionary, 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. 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 to determine if the key exists in the list.

The C# language uses the 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.

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

See Also	Add(Object, Object)Add(Object, Object) Performing Culture-Insensitive String Operations

ListDictionary.Keys ListDictionary.Keys

In this Article

Gets an ICollection containing the keys in the ListDictionary.

```
public System.Collections.ICollection Keys { get; }
```

```
member this.Keys : System.Collections.ICollection
```

Returns

ICollection ICollection

An ICollection containing the keys in the ListDictionary.

Examples

The following code example enumerates the elements of a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
```

```
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( ListDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Braeburn Apples           1.49
   1     Fuji Apples               1.29
   2     Gala Apples               1.49
   3     Golden Delicious Apples   1.29
   4     Granny Smith Apples       0.89
   5     Red Delicious Apples      0.99

*/
```

Remarks

The order of the values 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 ListDictionary. Therefore, changes to the ListDictionary continue to be reflected in the ICollection.

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

See Also	ICollectionICollection ValuesValues

ListDictionary ListDictionary

In this Article

Overloads

ListDictionary()	Creates an empty ListDictionary using the default comparer.
ListDictionary(IComparer) ListDictionary(IComparer)	Creates an empty ListDictionary using the specified comparer.

ListDictionary()

Creates an empty ListDictionary using the default comparer.

```
public ListDictionary ();
```

Examples

The following code example demonstrates several of the properties and methods of ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

      // Copies the ListDictionary to an array with DictionaryEntry elements.
      DictionaryEntry[] myArr = new DictionaryEntry[myCol.Count];
      myCol.CopyTo( myArr, 0 );

      // Displays the values in the array.
      Console.WriteLine( "Displays the elements in the array:" );
      Console.WriteLine( "   KEY                       VALUE" );
      for ( int i = 0; i < myArr.Length; i++ )
```

```
         Console.WriteLine( "   {0,-25} {1}", myArr[i].Key, myArr[i].Value );
      Console.WriteLine();

      // Searches for a key.
      if ( myCol.Contains( "Kiwis" ) )
         Console.WriteLine( "The collection contains the key \"Kiwis\"." );
      else
         Console.WriteLine( "The collection does not contain the key \"Kiwis\"." );
      Console.WriteLine();

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":"
);
      PrintKeysAndValues1( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues1( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( ListDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces output similar to the following.
Note that because a dictionary is implemented for fast keyed access the order
of the items in the dictionary are not gauranteed and, as a result, should not
be depended on.

Displays the elements using foreach:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
```

```
   Gala Apples              1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Braeburn Apples           1.49
   1     Fuji Apples               1.29
   2     Gala Apples               1.49
   3     Golden Delicious Apples   1.29
   4     Granny Smith Apples       0.89
   5     Red Delicious Apples      0.99

Displays the elements in the array:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

The collection does not contain the key "Kiwis".

The collection contains the following elements after removing "Plums":
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

The collection contains the following elements after it is cleared:
   KEY                       VALUE


*/
```

Remarks

The comparer determines whether two keys are equal. Every key in a ListDictionary must be unique. The default comparer is the key's implementation of Object.Equals.

This constructor is an O(1) operation.

See Also: Performing Culture-Insensitive String Operations

ListDictionary(IComparer) ListDictionary(IComparer)

Creates an empty ListDictionary using the specified comparer.

```
public ListDictionary (System.Collections.IComparer comparer);
```

```
new System.Collections.Specialized.ListDictionary : System.Collections.IComparer ->
System.Collections.Specialized.ListDictionary
```

Parameters

comparer IComparer IComparer

The IComparer 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).

Remarks

The comparer determines whether two keys are equal. Every key in a ListDictionary must be unique. The default comparer is the key's implementation of Object.Equals.

The custom comparer enables such scenarios as doing lookups with case-insensitive strings.

This constructor is an O(1) operation.

See Also

IComparerIComparer
Equals(Object)Equals(Object)
Performing Culture-Insensitive String Operations

ListDictionary.Remove ListDictionary.Remove

In this Article

Removes the entry with the specified key from the ListDictionary.

```
public void Remove (object key);
```

```
abstract member Remove : obj -> unit
override this.Remove : obj -> unit
```

Parameters

key Object Object

The key of the entry to remove.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example adds to and removes elements from a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Displays the values in the ListDictionary in three different ways.
      Console.WriteLine( "Initial contents of the ListDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes a key.
      myCol.Remove( "Plums" );
      Console.WriteLine( "The collection contains the following elements after removing \"Plums\":" 
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
```

```
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}


/*
This code produces the following output.

Initial contents of the ListDictionary:
   KEY                         VALUE
   Braeburn Apples             1.49
   Fuji Apples                 1.29
   Gala Apples                 1.49
   Golden Delicious Apples     1.29
   Granny Smith Apples         0.89
   Red Delicious Apples        0.99

The collection contains the following elements after removing "Plums":
   KEY                         VALUE
   Braeburn Apples             1.49
   Fuji Apples                 1.29
   Gala Apples                 1.49
   Golden Delicious Apples     1.29
   Granny Smith Apples         0.89
   Red Delicious Apples        0.99

The collection contains the following elements after it is cleared:
   KEY                         VALUE

*/
```

Remarks

If the ListDictionary does not contain an element with the specified key, the ListDictionary remains unchanged. No exception is thrown.

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

See Also

Add(Object, Object)Add(Object, Object)
Performing Culture-Insensitive String Operations

ListDictionary.SyncRoot ListDictionary.SyncRoot

In this Article

Gets an object that can be used to synchronize access to the ListDictionary.

```
public object SyncRoot { get; }
```

```
member this.SyncRoot : obj
```

Returns
Object Object

An object that can be used to synchronize access to the ListDictionary.

Examples

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration.

```
ListDictionary myCollection = new ListDictionary();
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 ListDictionary using the SyncRoot property. The synchronizing code must perform operations on the SyncRoot of the ListDictionary, not directly on the ListDictionary. 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 ListDictionary 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 IsSynchronizedIsSynchronized

ListDictionary.Values ListDictionary.Values

In this Article

Gets an ICollection containing the values in the ListDictionary.

```
public System.Collections.ICollection Values { get; }
```

```
member this.Values : System.Collections.ICollection
```

Returns
ICollection ICollection

An ICollection containing the values in the ListDictionary.

Examples

The following code example enumerates the elements of a ListDictionary.

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

public class SamplesListDictionary  {

   public static void Main()  {

      // Creates and initializes a new ListDictionary.
      ListDictionary myCol = new ListDictionary();
      myCol.Add( "Braeburn Apples", "1.49" );
      myCol.Add( "Fuji Apples", "1.29" );
      myCol.Add( "Gala Apples", "1.49" );
      myCol.Add( "Golden Delicious Apples", "1.29" );
      myCol.Add( "Granny Smith Apples", "0.89" );
      myCol.Add( "Red Delicious Apples", "0.99" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IDictionaryEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( IDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( IDictionary myCol )  {
```

```
      IDictionaryEnumerator myEnumerator = myCol.GetEnumerator();
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "   {0,-25} {1}", myEnumerator.Key, myEnumerator.Value );
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( ListDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the IDictionaryEnumerator:
   KEY                       VALUE
   Braeburn Apples           1.49
   Fuji Apples               1.29
   Gala Apples               1.49
   Golden Delicious Apples   1.29
   Granny Smith Apples       0.89
   Red Delicious Apples      0.99

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     Braeburn Apples           1.49
   1     Fuji Apples               1.29
   2     Gala Apples               1.49
   3     Golden Delicious Apples   1.29
   4     Granny Smith Apples       0.89
   5     Red Delicious Apples      0.99

*/
```

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 ListDictionary. Therefore, changes to the ListDictionary continue to be reflected in the ICollection.

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

See Also	ICollectionICollection KeysKeys

NameObjectCollectionBase NameObjectCollectionBase Class

Provides the `abstract` base class for a collection of associated String keys and Object values that can be accessed either with the key or with the index.

Declaration

```
[Serializable]
public abstract class NameObjectCollectionBase : System.Collections.ICollection,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

```
type NameObjectCollectionBase = class
    interface ICollection
    interface ISerializable
    interface IDeserializationCallback
    interface IEnumerable
```

Inheritance Hierarchy

`Object` `Object`

Remarks

The underlying structure for this class is a hash table.

Each element is a key/value pair.

The capacity of a NameObjectCollectionBase is the number of elements the NameObjectCollectionBase can hold. As elements are added to a NameObjectCollectionBase, the capacity is automatically increased as required through reallocation.

The hash code provider dispenses hash codes for keys in the NameObjectCollectionBase instance. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

In .NET Framework version 1.0, this class uses culture-sensitive string comparisons. However, in .NET Framework version 1.1 and later, this class uses CultureInfo.InvariantCulture when comparing strings. For more information about how culture affects comparisons and sorting, see Performing Culture-Insensitive String Operations.

`null` is allowed as a key or as a value.

Caution

The BaseGet method does not distinguish between `null` which is returned because the specified key is not found and `null` which is returned because the value associated with the key is `null`.

Constructors

`NameObjectCollectionBase()`

`NameObjectCollectionBase()`

Initializes a new instance of the NameObjectCollectionBase class that is empty.

`NameObjectCollectionBase(IEqualityComparer)`

`NameObjectCollectionBase(IEqualityComparer)`

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the default initial capacity, and uses the specified IEqualityComparer object.

`NameObjectCollectionBase(Int32)`

`NameObjectCollectionBase(Int32)`

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity, and uses the default hash code provider and the default comparer.

`NameObjectCollectionBase(IHashCodeProvider, IComparer)`

`NameObjectCollectionBase(IHashCodeProvider, IComparer)`

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the default initial capacity, and uses the specified hash code provider and the specified comparer.

`NameObjectCollectionBase(Int32, IEqualityComparer)`

`NameObjectCollectionBase(Int32, IEqualityComparer)`

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer object.

`NameObjectCollectionBase(SerializationInfo, StreamingContext)`

`NameObjectCollectionBase(SerializationInfo, StreamingContext)`

Initializes a new instance of the NameObjectCollectionBase class that is serializable and uses the specified SerializationInfo and StreamingContext.

`NameObjectCollectionBase(Int32, IHashCodeProvider, IComparer)`

`NameObjectCollectionBase(Int32, IHashCodeProvider, IComparer)`

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity and uses the specified hash code provider and the specified comparer.

Properties

`Count`

`Count`

Gets the number of key/value pairs contained in the NameObjectCollectionBase instance.

`IsReadOnly`

`IsReadOnly`

Gets or sets a value indicating whether the NameObjectCollectionBase instance is read-only.

Keys

Keys

Gets a NameObjectCollectionBase.KeysCollection instance that contains all the keys in the NameObjectCollectionBase instance.

Methods

`BaseAdd(String, Object)`

`BaseAdd(String, Object)`

Adds an entry with the specified key and value into the NameObjectCollectionBase instance.

`BaseClear()`

`BaseClear()`

Removes all entries from the NameObjectCollectionBase instance.

`BaseGet(Int32)`

`BaseGet(Int32)`

Gets the value of the entry at the specified index of the NameObjectCollectionBase instance.

`BaseGet(String)`

`BaseGet(String)`

Gets the value of the first entry with the specified key from the NameObjectCollectionBase instance.

`BaseGetAllKeys()`

`BaseGetAllKeys()`

Returns a String array that contains all the keys in the NameObjectCollectionBase instance.

`BaseGetAllValues()`

`BaseGetAllValues()`

Returns an Object array that contains all the values in the NameObjectCollectionBase instance.

`BaseGetAllValues(Type)`

`BaseGetAllValues(Type)`

Returns an array of the specified type that contains all the values in the NameObjectCollectionBase instance.

`BaseGetKey(Int32)`

`BaseGetKey(Int32)`

Gets the key of the entry at the specified index of the NameObjectCollectionBase instance.

`BaseHasKeys()`

`BaseHasKeys()`

Gets a value indicating whether the NameObjectCollectionBase instance contains entries whose keys are not `null`.

`BaseRemove(String)`

`BaseRemove(String)`

Removes the entries with the specified key from the NameObjectCollectionBase instance.

`BaseRemoveAt(Int32)`

`BaseRemoveAt(Int32)`

Removes the entry at the specified index of the NameObjectCollectionBase instance.

`BaseSet(Int32, Object)`

`BaseSet(Int32, Object)`

Sets the value of the entry at the specified index of the NameObjectCollectionBase instance.

`BaseSet(String, Object)`

`BaseSet(String, Object)`

Sets the value of the first entry with the specified key in the NameObjectCollectionBase instance, if found; otherwise, adds an entry with the specified key and value into the NameObjectCollectionBase instance.

`GetEnumerator()`

`GetEnumerator()`

Returns an enumerator that iterates through the NameObjectCollectionBase.

`GetObjectData(SerializationInfo, StreamingContext)`

`GetObjectData(SerializationInfo, StreamingContext)`

Implements the ISerializable interface and returns the data needed to serialize the NameObjectCollectionBase instance.

`OnDeserialization(Object)`

`OnDeserialization(Object)`

Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.

`ICollection.CopyTo(Array, Int32)`

`ICollection.CopyTo(Array, Int32)`

Copies the entire NameObjectCollectionBase 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 NameObjectCollectionBase object is synchronized (thread safe).

`ICollection.SyncRoot`

`ICollection.SyncRoot`

Gets an object that can be used to synchronize access to the NameObjectCollectionBase 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 NameObjectCollectionBase, but derived classes can create their own synchronized versions of the NameObjectCollectionBase 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

String String
String String

NameObjectCollectionBase.BaseAdd NameObjectCollectionBase.BaseAdd

In this Article

Adds an entry with the specified key and value into the NameObjectCollectionBase instance.

```
protected void BaseAdd (string name, object value);
```

```
member this.BaseAdd : string * obj -> unit
```

Parameters

name String String

The String key of the entry to add. The key can be `null`.

value Object Object

The Object value of the entry to add. The value can be `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Examples

The following code example uses BaseAdd to create a new NameObjectCollectionBase with elements from an IDictionary.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );

      // Displays the keys and values of the MyCollection instance.
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }

   }

}


/*
This code produces the following output.

[0] : red, apple
[1] : yellow, banana
[2] : green, pear

*/
```

Remarks

If Count already equals the capacity, the capacity of the NameObjectCollectionBase is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If Count is less than the capacity, 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.

NameObjectCollectionBase.BaseClear NameObjectCollectionBase.BaseClear

In this Article

Removes all entries from the NameObjectCollectionBase instance.

```
protected void BaseClear ();
```

```
member this.BaseClear : unit -> unit
```

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Examples

The following code example uses BaseClear to remove all elements from a NameObjectCollectionBase.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

   // Clears all the elements in the collection.
   public void Clear()  {
      this.BaseClear();
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
```

```
      PrintKeysAndValues( myCol );

      // Removes all elements from the collection.
      myCol.Clear();
      Console.WriteLine( "After clearing the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
After clearing the collection (Count = 0):

*/
```

Remarks

Count is set to zero, and references to other objects from elements of the collection are also released.

This method is an O(1) operation.

NameObjectCollectionBase.BaseGet NameObjectCollectionBase.BaseGet

In this Article

Overloads

BaseGet(Int32) BaseGet(Int32)	Gets the value of the entry at the specified index of the NameObjectCollectionBase instance.
BaseGet(String) BaseGet(String)	Gets the value of the first entry with the specified key from the NameObjectCollectionBase instance.

BaseGet(Int32) BaseGet(Int32)

Gets the value of the entry at the specified index of the NameObjectCollectionBase instance.

```
protected object BaseGet (int index);
```

```
member this.BaseGet : int -> obj
```

Parameters

index Int32 Int32

The zero-based index of the value to get.

Returns

Object Object

An Object that represents the value of the entry at the specified index.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Examples

The following code example uses BaseGetKey and BaseGet to get specific keys and values.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
```

```
   }

   // Gets or sets the value associated with the specified key.
   public Object this[ String key ]  {
      get  {
         return( this.BaseGet( key ) );
      }
      set  {
         this.BaseSet( key, value );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Gets specific keys and values.
      Console.WriteLine( "The key at index 0 is {0}.", myCol[0].Key );
      Console.WriteLine( "The value at index 0 is {0}.", myCol[0].Value );
      Console.WriteLine( "The value associated with the key \"green\" is {0}.", myCol["green"] );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
The key at index 0 is red.
The value at index 0 is apple.
The value associated with the key "green" is pear.

*/
```

Remarks

This method is an O(1) operation.

BaseGet(String) BaseGet(String)

Gets the value of the first entry with the specified key from the NameObjectCollectionBase instance.

```
protected object BaseGet (string name);
```

```
member this.BaseGet : string -> obj
```

Parameters

name String String

The String key of the entry to get. The key can be `null`.

Returns

Object Object

An Object that represents the value of the first entry with the specified key, if found; otherwise, `null`.

Examples

The following code example uses BaseGetKey and BaseGet to get specific keys and values.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Gets or sets the value associated with the specified key.
   public Object this[ String key ]  {
      get  {
         return( this.BaseGet( key ) );
      }
      set  {
         this.BaseSet( key, value );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {
```

```
   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Gets specific keys and values.
      Console.WriteLine( "The key at index 0 is {0}.", myCol[0].Key );
      Console.WriteLine( "The value at index 0 is {0}.", myCol[0].Value );
      Console.WriteLine( "The value associated with the key \"green\" is {0}.", myCol["green"] );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
The key at index 0 is red.
The value at index 0 is apple.
The value associated with the key "green" is pear.

*/
```

Remarks

If the collection contains multiple entries with the specified key, this method returns only the first entry. To get the values of subsequent entries with the same key, use the enumerator to iterate through the collection and compare the keys.

Caution

This method returns `null` in the following cases: 1) if the specified key is not found; and 2) if the specified key is found and its associated value is `null`. This method does not distinguish between the two cases.

This method is an O(1) operation.

See Also

GetEnumerator()GetEnumerator()

NameObjectCollectionBase.BaseGetAllKeys NameObjectCollectionBase.BaseGetAllKeys

In this Article

Returns a String array that contains all the keys in the NameObjectCollectionBase instance.

```
protected string[] BaseGetAllKeys ();
```

```
member this.BaseGetAllKeys : unit -> string[]
```

Returns

String[]

A String array that contains all the keys in the NameObjectCollectionBase instance.

Examples

The following code example uses BaseGetAllKeys and BaseGetAllValues to get an array of the keys or an array of the values.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

   // Gets a String array that contains all the keys in the collection.
   public String[] AllKeys  {
      get  {
         return( this.BaseGetAllKeys() );
      }
   }

   // Gets an Object array that contains all the values in the collection.
   public Array AllValues  {
      get  {
         return( this.BaseGetAllValues() );
      }
   }

   // Gets a String array that contains all the values in the collection.
   public String[] AllStringValues  {
      get  {
```

```
         return( (String[]) this.BaseGetAllValues( typeof(System.String) ) );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Displays the list of keys.
      Console.WriteLine( "The list of keys:" );
      foreach ( String s in myCol.AllKeys )  {
         Console.WriteLine( "   {0}", s );
      }

      // Displays the list of values of type Object.
      Console.WriteLine( "The list of values (Object):" );
      foreach ( Object o in myCol.AllValues )  {
         Console.WriteLine( "   {0}", o.ToString() );
      }

      // Displays the list of values of type String.
      Console.WriteLine( "The list of values (String):" );
      foreach ( String s in myCol.AllValues )  {
         Console.WriteLine( "   {0}", s );
      }

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
The list of keys:
   red
   yellow
   green
The list of values (Object):
   apple
   banana
   pear
The list of values (String):
   apple
```

```
    banana
    pear

*/
```

Remarks

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

NameObjectCollectionBase.BaseGetAllValues NameObjectCollectionBase.BaseGetAllValues

In this Article

Overloads

BaseGetAllValues() BaseGetAllValues()	Returns an Object array that contains all the values in the NameObjectCollectionBase instance.
BaseGetAllValues(Type) BaseGetAllValues(Type)	Returns an array of the specified type that contains all the values in the NameObjectCollectionBase instance.

BaseGetAllValues() BaseGetAllValues()

Returns an Object array that contains all the values in the NameObjectCollectionBase instance.

```
protected object[] BaseGetAllValues ();
```

```
member this.BaseGetAllValues : unit -> obj[]
```

Returns

Object[]

An Object array that contains all the values in the NameObjectCollectionBase instance.

Examples

The following code example uses BaseGetAllKeys and BaseGetAllValues to get an array of the keys or an array of the values.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

   // Gets a String array that contains all the keys in the collection.
   public String[] AllKeys  {
```

```
      get  {
         return( this.BaseGetAllKeys() );
      }
   }

   // Gets an Object array that contains all the values in the collection.
   public Array AllValues  {
      get  {
         return( this.BaseGetAllValues() );
      }
   }

   // Gets a String array that contains all the values in the collection.
   public String[] AllStringValues  {
      get  {
         return( (String[]) this.BaseGetAllValues( typeof(System.String) ) );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Displays the list of keys.
      Console.WriteLine( "The list of keys:" );
      foreach ( String s in myCol.AllKeys )  {
         Console.WriteLine( "   {0}", s );
      }

      // Displays the list of values of type Object.
      Console.WriteLine( "The list of values (Object):" );
      foreach ( Object o in myCol.AllValues )  {
         Console.WriteLine( "   {0}", o.ToString() );
      }

      // Displays the list of values of type String.
      Console.WriteLine( "The list of values (String):" );
      foreach ( String s in myCol.AllValues )  {
         Console.WriteLine( "   {0}", s );
      }

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.
```

```
Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
The list of keys:
   red
   yellow
   green
The list of values (Object):
   apple
   banana
   pear
The list of values (String):
   apple
   banana
   pear

*/
```

Remarks

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

BaseGetAllValues(Type) BaseGetAllValues(Type)

Returns an array of the specified type that contains all the values in the NameObjectCollectionBase instance.

```
protected object[] BaseGetAllValues (Type type);
```

```
member this.BaseGetAllValues : Type -> obj[]
```

Parameters

type Type Type

A Type that represents the type of array to return.

Returns
Object[]

An array of the specified type that contains all the values in the NameObjectCollectionBase instance.

Exceptions
ArgumentNullException ArgumentNullException

`type` is `null`.

ArgumentException ArgumentException

`type` is not a valid Type.

Remarks

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

See Also TypeType

NameObjectCollectionBase.BaseGetKey NameObjectCollectionBase.BaseGetKey

In this Article

Gets the key of the entry at the specified index of the NameObjectCollectionBase instance.

```
protected string BaseGetKey (int index);
```

```
member this.BaseGetKey : int -> string
```

Parameters

index Int32 Int32

The zero-based index of the key to get.

Returns

String String

A String that represents the key of the entry at the specified index.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Examples

The following code example uses BaseGetKey and BaseGet to get specific keys and values.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Gets or sets the value associated with the specified key.
   public Object this[ String key ]  {
      get  {
         return( this.BaseGet( key ) );
      }
      set  {
         this.BaseSet( key, value );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
```

```
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Gets specific keys and values.
      Console.WriteLine( "The key at index 0 is {0}.", myCol[0].Key );
      Console.WriteLine( "The value at index 0 is {0}.", myCol[0].Value );
      Console.WriteLine( "The value associated with the key \"green\" is {0}.", myCol["green"] );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
The key at index 0 is red.
The value at index 0 is apple.
The value associated with the key "green" is pear.

*/
```

Remarks

This method is an O(1) operation.

NameObjectCollectionBase.BaseHasKeys NameObjectCollectionBase.BaseHasKeys

In this Article

Gets a value indicating whether the NameObjectCollectionBase instance contains entries whose keys are not `null`.

```
protected bool BaseHasKeys ();
```

```
member this.BaseHasKeys : unit -> bool
```

Returns

Boolean Boolean

`true` if the NameObjectCollectionBase instance contains entries whose keys are not `null`; otherwise, `false`.

Examples

The following code example uses BaseHasKeys to determine if the collection contains keys that are not `null`.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Creates an empty collection.
   public MyCollection()  {
   }

   // Adds an entry to the collection.
   public void Add( String key, Object value )  {
      this.BaseAdd( key, value );
   }

   // Gets a value indicating whether the collection contains keys that are not a null reference.
   public Boolean HasKeys  {
      get  {
         return( this.BaseHasKeys() );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates an empty MyCollection instance.
      MyCollection myCol = new MyCollection();
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
```

```
      PrintKeysAndValues( myCol );
      Console.WriteLine( "HasKeys? {0}", myCol.HasKeys );

      Console.WriteLine();

      // Adds an item to the collection.
      myCol.Add( "blue", "sky" );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );
      Console.WriteLine( "HasKeys? {0}", myCol.HasKeys );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 0):
HasKeys? False

Initial state of the collection (Count = 1):
[0] : blue, sky
HasKeys? True

*/
```

Remarks

This method is an O(1) operation.

NameObjectCollectionBase.BaseRemove NameObjectCollectionBase.BaseRemove

In this Article

Removes the entries with the specified key from the NameObjectCollectionBase instance.

```
protected void BaseRemove (string name);
```

```
member this.BaseRemove : string -> unit
```

Parameters

name String String

The String key of the entries to remove. The key can be `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Examples

The following code example uses BaseRemove and BaseRemoveAt to remove elements from a NameObjectCollectionBase.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

   // Removes an entry with the specified key from the collection.
   public void Remove( String key )  {
      this.BaseRemove( key );
   }

   // Removes an entry in the specified index from the collection.
   public void Remove( int index )  {
      this.BaseRemoveAt( index );
   }

}
```

```
}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Removes an element at a specific index.
      myCol.Remove( 1 );
      Console.WriteLine( "After removing the element at index 1 (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Removes an element with a specific key.
      myCol.Remove( "red" );
      Console.WriteLine( "After removing the element with the key \"red\" (Count = {0}):",
myCol.Count );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
After removing the element at index 1 (Count = 2):
[0] : red, apple
[1] : green, pear
After removing the element with the key "red" (Count = 1):
[0] : green, pear

*/
```

Remarks

If the NameObjectCollectionBase does not contain an element with the specified key, the NameObjectCollectionBase 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 Also	Performing Culture-Insensitive String Operations

NameObjectCollectionBase.BaseRemoveAt NameObjectCollectionBase.BaseRemoveAt

In this Article

Removes the entry at the specified index of the NameObjectCollectionBase instance.

```
protected void BaseRemoveAt (int index);
```

```
member this.BaseRemoveAt : int -> unit
```

Parameters

index Int32 Int32

The zero-based index of the entry to remove.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

NotSupportedException NotSupportedException

The collection is read-only.

Examples

The following code example uses BaseRemove and BaseRemoveAt to remove elements from a NameObjectCollectionBase.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

   // Removes an entry with the specified key from the collection.
   public void Remove( String key )  {
      this.BaseRemove( key );
   }

   // Removes an entry in the specified index from the collection.
   public void Remove( int index )  {
```

```
   public void Remove( int index )  {
      this.BaseRemoveAt( index );
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Removes an element at a specific index.
      myCol.Remove( 1 );
      Console.WriteLine( "After removing the element at index 1 (Count = {0}):", myCol.Count );
      PrintKeysAndValues( myCol );

      // Removes an element with a specific key.
      myCol.Remove( "red" );
      Console.WriteLine( "After removing the element with the key \"red\" (Count = {0}):",
myCol.Count );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection (Count = 3):
[0] : red, apple
[1] : yellow, banana
[2] : green, pear
After removing the element at index 1 (Count = 2):
[0] : red, apple
[1] : green, pear
After removing the element with the key "red" (Count = 1):
[0] : green, pear

*/
```

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.

NameObjectCollectionBase.BaseSet NameObjectCollectionBase.BaseSet

In this Article

Overloads

BaseSet(Int32, Object) BaseSet(Int32, Object)	Sets the value of the entry at the specified index of the NameObjectCollectionBase instance.
BaseSet(String, Object) BaseSet(String, Object)	Sets the value of the first entry with the specified key in the NameObjectCollectionBase instance, if found; otherwise, adds an entry with the specified key and value into the NameObjectCollectionBase instance.

BaseSet(Int32, Object) BaseSet(Int32, Object)

Sets the value of the entry at the specified index of the NameObjectCollectionBase instance.

```
protected void BaseSet (int index, object value);
```

```
member this.BaseSet : int * obj -> unit
```

Parameters

index Int32 Int32

The zero-based index of the entry to set.

value Object Object

The Object that represents the new value of the entry to set. The value can be `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Examples

The following code example uses BaseSet to set the value of a specific element.

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

public class MyCollection : NameObjectCollectionBase  {

   // Gets or sets the value at the specified index.
   public Object this[ int index ]  {
      get  {
         return( this.BaseGet( index ) );
      }
```

```
      }
      set  {
         this.BaseSet( index, value );
      }
   }

   // Gets or sets the value associated with the specified key.
   public Object this[ String key ]  {
      get  {
         return( this.BaseGet( key ) );
      }
      set  {
         this.BaseSet( key, value );
      }
   }

   // Gets a String array that contains all the keys in the collection.
   public String[] AllKeys  {
      get  {
         return( this.BaseGetAllKeys() );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection:" );
      PrintKeysAndValues2( myCol );
      Console.WriteLine();

      // Sets the value at index 1.
      myCol[1] = "sunflower";
      Console.WriteLine( "After setting the value at index 1:" );
      PrintKeysAndValues2( myCol );
      Console.WriteLine();

      // Sets the value associated with the key "red".
      myCol["red"] = "tulip";
      Console.WriteLine( "After setting the value associated with the key \"red\":" );
      PrintKeysAndValues2( myCol );

   }

   public static void PrintKeysAndValues2( MyCollection myCol )  {
      foreach ( String s in myCol.AllKeys )  {
         Console.WriteLine( "{0}, {1}", s, myCol[s] );
      }
   }

}
```

```
/*
This code produces the following output.

Initial state of the collection:
red, apple
yellow, banana
green, pear

After setting the value at index 1:
red, apple
yellow, sunflower
green, pear

After setting the value associated with the key "red":
red, tulip
yellow, sunflower
green, pear

*/
```

Remarks

This method is an O(1) operation.

See Also: Performing Culture-Insensitive String Operations

BaseSet(String, Object) BaseSet(String, Object)

Sets the value of the first entry with the specified key in the NameObjectCollectionBase instance, if found; otherwise, adds an entry with the specified key and value into the NameObjectCollectionBase instance.

```
protected void BaseSet (string name, object value);
```

```
member this.BaseSet : string * obj -> unit
```

Parameters

name: String String

The String key of the entry to set. The key can be `null`.

value: Object Object

The Object that represents the new value of the entry to set. The value can be `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Examples

The following code example uses BaseSet to set the value of a specific element.

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

public class MyCollection : NameObjectCollectionBase  {
```

```
   // Gets or sets the value at the specified index.
   public Object this[ int index ]  {
      get  {
         return( this.BaseGet( index ) );
      }
      set  {
         this.BaseSet( index, value );
      }
   }

   // Gets or sets the value associated with the specified key.
   public Object this[ String key ]  {
      get  {
         return( this.BaseGet( key ) );
      }
      set  {
         this.BaseSet( key, value );
      }
   }

   // Gets a String array that contains all the keys in the collection.
   public String[] AllKeys  {
      get  {
         return( this.BaseGetAllKeys() );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection instance.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myCol = new MyCollection( d );
      Console.WriteLine( "Initial state of the collection:" );
      PrintKeysAndValues2( myCol );
      Console.WriteLine();

      // Sets the value at index 1.
      myCol[1] = "sunflower";
      Console.WriteLine( "After setting the value at index 1:" );
      PrintKeysAndValues2( myCol );
      Console.WriteLine();

      // Sets the value associated with the key "red".
      myCol["red"] = "tulip";
      Console.WriteLine( "After setting the value associated with the key \"red\":" );
      PrintKeysAndValues2( myCol );

   }

   public static void PrintKeysAndValues2( MyCollection myCol )  {
      foreach ( String s in myCol.AllKeys )  {
         Console.WriteLine( "{0}, {1}", s, myCol[s] );
```

```
         Console.WriteLine( "{0}, {1}", s, myCol[s] );
      }
   }

}


/*
This code produces the following output.

Initial state of the collection:
red, apple
yellow, banana
green, pear

After setting the value at index 1:
red, apple
yellow, sunflower
green, pear

After setting the value associated with the key "red":
red, tulip
yellow, sunflower
green, pear

*/
```

Remarks

If the collection contains multiple entries with the specified key, this method sets only the first entry. To set the values of subsequent entries with the same key, use the enumerator to iterate through the collection and compare the keys.

This method is an O(1) operation.

See Also

GetEnumerator()GetEnumerator()

Performing Culture-Insensitive String Operations

NameObjectCollectionBase.Count NameObjectCollectionBase.Count

In this Article

Gets the number of key/value pairs contained in the NameObjectCollectionBase instance.

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

```
member this.Count : int
```

Returns
Int32 Int32

The number of key/value pairs contained in the NameObjectCollectionBase instance.

Remarks

The capacity is the number of elements that the NameObjectCollectionBase can store. Count is the number of elements that are actually in the NameObjectCollectionBase.

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.

NameObjectCollectionBase.GetEnumerator NameObjectCollectionBase.GetEnumerator

In this Article

Returns an enumerator that iterates through the NameObjectCollectionBase.

```
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 NameObjectCollectionBase instance.

Remarks

This enumerator returns the keys of the collection as strings.

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.

NameObjectCollectionBase.GetObjectData NameObjectCollectionBase.GetObjectData

In this Article

Implements the ISerializable interface and returns the data needed to serialize the NameObjectCollectionBase instance.

```
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 that contains the information required to serialize the NameObjectCollectionBase instance.

context StreamingContext StreamingContext

A StreamingContext object that contains the source and destination of the serialized stream associated with the NameObjectCollectionBase instance.

Exceptions

ArgumentNullException ArgumentNullException

`info` is `null`.

Remarks

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

See Also

ISerializableISerializable
SerializationInfoSerializationInfo
StreamingContextStreamingContext

NameObjectCollectionBase.ICollection.CopyTo

In this Article

Copies the entire NameObjectCollectionBase 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 NameObjectCollectionBase. 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 NameObjectCollectionBase is greater than the available space from `index` to the end of the destination `array`.

InvalidCastException

The type of the source NameObjectCollectionBase 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.

While the ICollection.CopyTo method is not visible to COM clients by default, inheriting the NameObjectCollectionBase class can expose it and can cause undesirable behavior in COM clients.

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

NameObjectCollectionBase.ICollection.IsSynchronized

In this Article

Gets a value indicating whether access to the NameObjectCollectionBase object is synchronized (thread safe).

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

Returns
Boolean

`true` if access to the NameObjectCollectionBase object is synchronized (thread safe); otherwise, `false`. The default is `false`.

Remarks

A NameObjectCollectionBase object is not synchronized. Derived classes can provide a synchronized version of the NameObjectCollectionBase 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 property during the entire enumeration.

```
// Create a collection derived from NameObjectCollectionBase
ICollection myCollection = new DerivedCollection();
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

NameObjectCollectionBase.ICollection.SyncRoot

In this Article

Gets an object that can be used to synchronize access to the NameObjectCollectionBase object.

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

Returns
Object

An object that can be used to synchronize access to the NameObjectCollectionBase object.

Remarks

Derived classes can provide their own synchronized version of the NameObjectCollectionBase class using the SyncRoot property. The synchronizing code must perform operations on the SyncRoot property of the NameObjectCollectionBase object, not directly on the NameObjectCollectionBase 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 NameObjectCollectionBase 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.

```
// Create a collection derived from NameObjectCollectionBase
ICollection myCollection = new DerivedCollection();
lock(myCollection.SyncRoot)
{
    foreach (object item in myCollection)
    {
        // Insert your code here.
    }
}
```

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

NameObjectCollectionBase.IsReadOnly NameObjectCollectionBase.IsReadOnly

In this Article

Gets or sets a value indicating whether the NameObjectCollectionBase instance is read-only.

```
protected bool IsReadOnly { get; set; }
```

```
member this.IsReadOnly : bool with get, set
```

Returns
Boolean Boolean

`true` if the NameObjectCollectionBase instance is read-only; otherwise, `false`.

Examples

The following code example creates a read-only collection.

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

public class MyCollection : NameObjectCollectionBase  {

   private DictionaryEntry _de = new DictionaryEntry();

   // Gets a key-and-value pair (DictionaryEntry) using an index.
   public DictionaryEntry this[ int index ]  {
      get  {
         _de.Key = this.BaseGetKey( index );
         _de.Value = this.BaseGet( index );
         return( _de );
      }
   }

   // Adds elements from an IDictionary into the new collection.
   public MyCollection( IDictionary d, Boolean bReadOnly )  {
      foreach ( DictionaryEntry de in d )  {
         this.BaseAdd( (String) de.Key, de.Value );
      }
      this.IsReadOnly = bReadOnly;
   }

   // Adds an entry to the collection.
   public void Add( String key, Object value )  {
      this.BaseAdd( key, value );
   }

}

public class SamplesNameObjectCollectionBase  {

   public static void Main()  {

      // Creates and initializes a new MyCollection that is read-only.
      IDictionary d = new ListDictionary();
      d.Add( "red", "apple" );
      d.Add( "yellow", "banana" );
      d.Add( "green", "pear" );
      MyCollection myROCol = new MyCollection( d, true );
```

```
      // Tries to add a new item.
      try  {
         myROCol.Add( "blue", "sky" );
      }
      catch ( NotSupportedException e )  {
         Console.WriteLine( e.ToString() );
      }

      // Displays the keys and values of the MyCollection.
      Console.WriteLine( "Read-Only Collection:" );
      PrintKeysAndValues( myROCol );

   }

   public static void PrintKeysAndValues( MyCollection myCol )  {
      for ( int i = 0; i < myCol.Count; i++ )  {
         Console.WriteLine( "[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value );
      }
   }

}


/*
This code produces the following output.

System.NotSupportedException: Collection is read-only.
   at System.Collections.Specialized.NameObjectCollectionBase.BaseAdd(String name, Object value)
   at SamplesNameObjectCollectionBase.Main()
Read-Only Collection:
[0] : red, apple
[1] : yellow, banana
[2] : green, pear

*/
```

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.

NameObjectCollectionBase.Keys NameObjectCollectionBase.Keys

In this Article

Gets a NameObjectCollectionBase.KeysCollection instance that contains all the keys in the NameObjectCollectionBase instance.

```
public virtual System.Collections.Specialized.NameObjectCollectionBase.KeysCollection Keys { get; }
```

```
member this.Keys : System.Collections.Specialized.NameObjectCollectionBase.KeysCollection
```

Returns
NameObjectCollectionBase.KeysCollection NameObjectCollectionBase.KeysCollection

A NameObjectCollectionBase.KeysCollection instance that contains all the keys in the NameObjectCollectionBase instance.

Remarks

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

NameObjectCollectionBase NameObjectCollectionBase

In this Article

Overloads

NameObjectCollectionBase()	Initializes a new instance of the NameObjectCollectionBase class that is empty.
NameObjectCollectionBase(IEqualityComparer) NameObjectCollectionBase(IEqualityComparer)	Initializes a new instance of the NameObjectCollectionBase class that is empty, has the default initial capacity, and uses the specified IEqualityComparer object.
NameObjectCollectionBase(Int32) NameObjectCollectionBase(Int32)	Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity, and uses the default hash code provider and the default comparer.
NameObjectCollectionBase(IHashCodeProvider, IComparer) NameObjectCollectionBase(IHashCodeProvider, IComparer)	Initializes a new instance of the NameObjectCollectionBase class that is empty, has the default initial capacity, and uses the specified hash code provider and the specified comparer.
NameObjectCollectionBase(Int32, IEqualityComparer) NameObjectCollectionBase(Int32, IEqualityComparer)	Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer object.
NameObjectCollectionBase(SerializationInfo, StreamingContext) NameObjectCollectionBase(SerializationInfo, StreamingContext)	Initializes a new instance of the NameObjectCollectionBase class that is serializable and uses the specified SerializationInfo and StreamingContext.
NameObjectCollectionBase(Int32, IHashCodeProvider, IComparer) NameObjectCollectionBase(Int32, IHashCodeProvider, IComparer)	Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity and uses the specified hash code provider and the specified comparer.

NameObjectCollectionBase()

Initializes a new instance of the NameObjectCollectionBase class that is empty.

```
protected NameObjectCollectionBase ();
```

Remarks

The capacity of a NameObjectCollectionBase is the number of elements that the NameObjectCollectionBase can hold. As elements are added to a NameObjectCollectionBase, 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 NameObjectCollectionBase.

The hash code provider dispenses hash codes for keys in the NameObjectCollectionBase instance. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

This constructor is an O(1) operation.

See Also: Performing Culture-Insensitive String Operations

NameObjectCollectionBase(IEqualityComparer) NameObjectCollectionBase(IEqualityComparer)

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the default initial capacity, and uses the specified IEqualityComparer object.

```
protected NameObjectCollectionBase (System.Collections.IEqualityComparer equalityComparer);
```

```
new System.Collections.Specialized.NameObjectCollectionBase : System.Collections.IEqualityComparer -
> System.Collections.Specialized.NameObjectCollectionBase
```

Parameters

equalityComparer IEqualityComparer IEqualityComparer

The IEqualityComparer object to use to determine whether two keys are equal and to generate hash codes for the keys in the collection.

Remarks

The capacity of a NameObjectCollectionBase object is the number of elements that the NameObjectCollectionBase can hold. As elements are added to a NameObjectCollectionBase, 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 NameObjectCollectionBase.

The IEqualityComparer object combines the comparer and the hash code provider. The hash code provider dispenses hash codes for keys in the NameObjectCollectionBase. The comparer determines whether two keys are equal.

This constructor is an O(1) operation.

See Also: IEqualityComparerIEqualityComparer
Performing Culture-Insensitive String Operations

NameObjectCollectionBase(Int32) NameObjectCollectionBase(Int32)

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity, and uses the default hash code provider and the default comparer.

```
protected NameObjectCollectionBase (int capacity);
```

```
new System.Collections.Specialized.NameObjectCollectionBase : int ->
System.Collections.Specialized.NameObjectCollectionBase
```

Parameters

capacity Int32 Int32

The approximate number of entries that the NameObjectCollectionBase instance can initially contain.

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

Remarks

The capacity of a NameObjectCollectionBase is the number of elements that the NameObjectCollectionBase can hold. As elements are added to a NameObjectCollectionBase, 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 NameObjectCollectionBase.

The hash code provider dispenses hash codes for keys in the NameObjectCollectionBase instance. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

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

See Also Performing Culture-Insensitive String Operations

NameObjectCollectionBase(IHashCodeProvider, IComparer) NameObjectCollectionBase(IHashCodeProvider, IComparer)

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the default initial capacity, and uses the specified hash code provider and the specified comparer.

```
[System.Obsolete("Please use NameObjectCollectionBase(IEqualityComparer) instead.")]
protected NameObjectCollectionBase (System.Collections.IHashCodeProvider hashProvider,
System.Collections.IComparer comparer);
```

```
new System.Collections.Specialized.NameObjectCollectionBase : System.Collections.IHashCodeProvider *
System.Collections.IComparer -> System.Collections.Specialized.NameObjectCollectionBase
```

Parameters

hashProvider IHashCodeProvider IHashCodeProvider

The IHashCodeProvider that will supply the hash codes for all keys in the NameObjectCollectionBase instance.

comparer IComparer IComparer

The IComparer to use to determine whether two keys are equal.

Attributes ObsoleteAttribute

Remarks

The capacity of a NameObjectCollectionBase is the number of elements that the NameObjectCollectionBase can hold. As elements are added to a NameObjectCollectionBase, 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 NameObjectCollectionBase.

The hash code provider dispenses hash codes for keys in the NameObjectCollectionBase instance. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

This constructor is an O(1) operation.

See Also
IHashCodeProviderIHashCodeProvider
IComparerIComparer
Performing Culture-Insensitive String Operations

NameObjectCollectionBase(Int32, IEqualityComparer) NameObjectCollectionBase(Int32, IEqualityComparer)

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer object.

```
protected NameObjectCollectionBase (int capacity, System.Collections.IEqualityComparer equalityComparer);
```

```
new System.Collections.Specialized.NameObjectCollectionBase : int * System.Collections.IEqualityComparer -> System.Collections.Specialized.NameObjectCollectionBase
```

Parameters

capacity Int32 Int32

The approximate number of entries that the NameObjectCollectionBase object can initially contain.

equalityComparer IEqualityComparer IEqualityComparer

The IEqualityComparer object to use to determine whether two keys are equal and to generate hash codes for the keys in the collection.

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

Remarks

The capacity of a NameObjectCollectionBase object is the number of elements that the NameObjectCollectionBase can hold. As elements are added to a NameObjectCollectionBase, 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 NameObjectCollectionBase.

The IEqualityComparer object combines the comparer and the hash code provider. The hash code provider dispenses hash codes for keys in the NameObjectCollectionBase. The comparer determines whether two keys are equal.

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

See Also
IEqualityComparerIEqualityComparer
Performing Culture-Insensitive String Operations

NameObjectCollectionBase(SerializationInfo, StreamingContext) NameObjectCollectionBase(SerializationInfo, StreamingContext)

Initializes a new instance of the NameObjectCollectionBase class that is serializable and uses the specified SerializationInfo and StreamingContext.

```
protected NameObjectCollectionBase (System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);
```

```
new System.Collections.Specialized.NameObjectCollectionBase :
System.Runtime.Serialization.SerializationInfo * System.Runtime.Serialization.StreamingContext ->
System.Collections.Specialized.NameObjectCollectionBase
```

Parameters

info SerializationInfo SerializationInfo

A SerializationInfo object that contains the information required to serialize the new NameObjectCollectionBase instance.

context StreamingContext StreamingContext

A StreamingContext object that contains the source and destination of the serialized stream associated with the new NameObjectCollectionBase instance.

Remarks

This constructor is an O(1) operation.

See Also

ISerializableISerializable
SerializationInfoSerializationInfo
StreamingContextStreamingContext
Performing Culture-Insensitive String Operations

NameObjectCollectionBase(Int32, IHashCodeProvider, IComparer) NameObjectCollectionBase(Int32, IHashCodeProvider, IComparer)

Initializes a new instance of the NameObjectCollectionBase class that is empty, has the specified initial capacity and uses the specified hash code provider and the specified comparer.

```
[System.Obsolete("Please use NameObjectCollectionBase(Int32, IEqualityComparer) instead.")]
protected NameObjectCollectionBase (int capacity, System.Collections.IHashCodeProvider hashProvider,
System.Collections.IComparer comparer);
```

```
new System.Collections.Specialized.NameObjectCollectionBase : int *
System.Collections.IHashCodeProvider * System.Collections.IComparer ->
System.Collections.Specialized.NameObjectCollectionBase
```

Parameters

capacity Int32 Int32

The approximate number of entries that the NameObjectCollectionBase instance can initially contain.

hashProvider IHashCodeProvider IHashCodeProvider

The IHashCodeProvider that will supply the hash codes for all keys in the NameObjectCollectionBase instance.

comparer IComparer IComparer

The IComparer to use to determine whether two keys are equal.

Attributes ObsoleteAttribute

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

Remarks

The capacity of a NameObjectCollectionBase is the number of elements that the NameObjectCollectionBase can hold. As elements are added to a NameObjectCollectionBase, 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 NameObjectCollectionBase.

The hash code provider dispenses hash codes for keys in the NameObjectCollectionBase instance. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

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

See Also

IHashCodeProviderIHashCodeProvider
IComparerIComparer
Performing Culture-Insensitive String Operations

NameObjectCollectionBase.OnDeserialization NameObjectCollectionBase.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 Object Object

The source of the deserialization event.

Exceptions

SerializationException SerializationException

The SerializationInfo object associated with the current NameObjectCollectionBase instance is invalid.

Remarks

While the OnDeserialization method is not visible to COM clients by default, inheriting the NameObjectCollectionBase class can expose it and can cause undesirable behavior in COM clients.

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

See Also

ISerializableISerializable

GetObjectData(SerializationInfo, StreamingContext)GetObjectData(SerializationInfo, StreamingContext)

NameObjectCollectionBase.KeysCollection NameObjectCollectionBase.KeysCollection Class

Represents a collection of the String keys of a collection.

Declaration

```
[Serializable]
public class NameObjectCollectionBase.KeysCollection : System.Collections.ICollection
```

```
type NameObjectCollectionBase.KeysCollection = class
    interface ICollection
    interface IEnumerable
```

Inheritance Hierarchy

`Object` `Object`

Properties

Count

Count

Gets the number of keys in the NameObjectCollectionBase.KeysCollection.

Item[Int32]

Item[Int32]

Gets the entry at the specified index of the collection.

Methods

Get(Int32)

Get(Int32)

Gets the key at the specified index of the collection.

GetEnumerator()

GetEnumerator()

Returns an enumerator that iterates through the NameObjectCollectionBase.KeysCollection.

ICollection.CopyTo(Array, Int32)

ICollection.CopyTo(Array, Int32)

Copies the entire NameObjectCollectionBase.KeysCollection 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 NameObjectCollectionBase.KeysCollection is synchronized (thread safe).

`ICollection.SyncRoot`

`ICollection.SyncRoot`

Gets an object that can be used to synchronize access to the NameObjectCollectionBase.KeysCollection.

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 NameObjectCollectionBase.KeysCollection, but derived classes can create their own synchronized versions of the NameObjectCollectionBase.KeysCollection 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.

NameObjectCollectionBase.KeysCollection.Count NameObjectCollectionBase.KeysCollection.Count

In this Article

Gets the number of keys in the NameObjectCollectionBase.KeysCollection.

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

```
member this.Count : int
```

Returns
Int32 Int32

The number of keys in the NameObjectCollectionBase.KeysCollection.

Remarks

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

NameObjectCollectionBase.KeysCollection.Get NameObjectCollectionBase.KeysCollection.Get

In this Article

Gets the key at the specified index of the collection.

```
public virtual string Get (int index);
```

```
abstract member Get : int -> string
override this.Get : int -> string
```

Parameters

index Int32 Int32

The zero-based index of the key to get from the collection.

Returns

String String

A String that contains the key at the specified index of the collection.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Remarks

This method is an O(1) operation.

NameObjectCollectionBase.KeysCollection.GetEnumerator NameObjectCollectionBase.KeysCollection.GetEnumerator

In this Article

Returns an enumerator that iterates through the NameObjectCollectionBase.KeysCollection.

```
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 NameObjectCollectionBase.KeysCollection.

Remarks

This enumerator returns the keys of the collection as strings.

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.

NameObjectCollectionBase.KeysCollection.ICollection.CopyTo

In this Article

Copies the entire NameObjectCollectionBase.KeysCollection 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 NameObjectCollectionBase.KeysCollection. 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 NameObjectCollectionBase.KeysCollection is greater than the available space from `index` to the end of the destination `array`.

InvalidCastException

The type of the source NameObjectCollectionBase.KeysCollection 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.

NameObjectCollectionBase.KeysCollection.ICollection.IsSynchronized

In this Article

Gets a value indicating whether access to the NameObjectCollectionBase.KeysCollection is synchronized (thread safe).

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

Returns
Boolean

`true` if access to the NameObjectCollectionBase.KeysCollection 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.

```
// Create a collection derived from NameObjectCollectionBase
NameObjectCollectionBase myBaseCollection = new DerivedCollection();
// Get the ICollection from NameObjectCollectionBase.KeysCollection
ICollection myKeysCollection = myBaseCollection.Keys;
lock(myKeysCollection.SyncRoot)
{
    foreach (object item in myKeysCollection)
    {
        // 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 NameObjectCollectionBase.KeysCollection using the SyncRoot property. The synchronizing code must perform operations on the SyncRoot of the NameObjectCollectionBase.KeysCollection, not directly on the NameObjectCollectionBase.KeysCollection. 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 NameObjectCollectionBase.KeysCollection 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.

NameObjectCollectionBase.KeysCollection.ICollection.SyncRoot

In this Article

Gets an object that can be used to synchronize access to the NameObjectCollectionBase.KeysCollection.

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

Returns
Object

An object that can be used to synchronize access to the NameObjectCollectionBase.KeysCollection.

Examples

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration.

```
// Create a collection derived from NameObjectCollectionBase
NameObjectCollectionBase myBaseCollection = new DerivedCollection();
// Get the ICollection from NameObjectCollectionBase.KeysCollection
ICollection myKeysCollection = myBaseCollection.Keys;
lock(myKeysCollection.SyncRoot)
{
    foreach (object item in myKeysCollection)
    {
        // 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 NameObjectCollectionBase.KeysCollection using the SyncRoot property. The synchronizing code must perform operations on the SyncRoot of the NameObjectCollectionBase.KeysCollection, not directly on the NameObjectCollectionBase.KeysCollection. 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 NameObjectCollectionBase.KeysCollection 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.

NameObjectCollectionBase.KeysCollection.Item[Int32] NameObjectCollectionBase.KeysCollection.Item[Int32]

In this Article

Gets the entry at the specified index of the collection.

```
public string this[int index] { get; }
```

```
member this.Item(int) : string
```

Parameters

index Int32 Int32

The zero-based index of the entry to locate in the collection.

Returns

String String

The String key of the entry at the specified index of the collection.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Remarks

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[index]` (In Visual Basic, `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.

NameValueCollection NameValueCollection Class

Represents a collection of associated String keys and String values that can be accessed either with the key or with the index.

Declaration

```
[Serializable]
public class NameValueCollection : System.Collections.Specialized.NameObjectCollectionBase
```

```
type NameValueCollection = class
    inherit NameObjectCollectionBase
```

Inheritance Hierarchy

`Object` `Object`

`NameObjectCollectionBase` `NameObjectCollectionBase`

Remarks

This collection is based on the NameObjectCollectionBase class. Each element of the collection is a key/value pair. However, unlike the NameObjectCollectionBase, this class can store multiple string values under a single key.

This class can be used for headers, query strings and form data.

Collections of this type do not preserve the ordering of elements, and no particular ordering is guaranteed when enumerating the collection.

The capacity of a NameValueCollection is the number of elements the NameValueCollection can hold. As elements are added, its capacity is automatically increased as required through reallocation.

The hash code provider dispenses hash codes for keys in the NameValueCollection. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is a CaseInsensitiveComparer that uses the conventions of the invariant culture; that is, key comparisons are case-insensitive by default. To perform case-sensitive key comparisons, call the NameValueCollection.NameValueCollection(IEqualityComparer) constructor, and provide a value of StringComparer.CurrentCulture, StringComparer.InvariantCulture, or StringComparer.Ordinal as the `equalityComparer` argument. For more information about how culture affects comparisons and sorting, see Performing Culture-Insensitive String Operations.

`null` is allowed as a key or as a value.

Caution

The Get method does not distinguish between `null` which is returned because the specified key is not found and `null` which is returned because the value associated with the key is `null`.

Constructors

`NameValueCollection()`

`NameValueCollection()`

Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity and uses the default case-insensitive hash code provider and the default case-insensitive comparer.

`NameValueCollection(IEqualityComparer)`

`NameValueCollection(IEqualityComparer)`

Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity, and uses the specified IEqualityComparer object.

`NameValueCollection(NameValueCollection)`

`NameValueCollection(NameValueCollection)`

Copies the entries from the specified NameValueCollection to a new NameValueCollection with the same initial capacity as the number of entries copied and using the same hash code provider and the same comparer as the source collection.

`NameValueCollection(Int32)`

`NameValueCollection(Int32)`

Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity and uses the default case-insensitive hash code provider and the default case-insensitive comparer.

`NameValueCollection(IHashCodeProvider, IComparer)`

`NameValueCollection(IHashCodeProvider, IComparer)`

Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity and uses the specified hash code provider and the specified comparer.

`NameValueCollection(Int32, IEqualityComparer)`

`NameValueCollection(Int32, IEqualityComparer)`

Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer object.

`NameValueCollection(Int32, NameValueCollection)`

`NameValueCollection(Int32, NameValueCollection)`

Copies the entries from the specified NameValueCollection to a new NameValueCollection with the specified initial capacity or the same initial capacity as the number of entries copied, whichever is greater, and using the default case-insensitive hash code provider and the default case-insensitive comparer.

`NameValueCollection(SerializationInfo, StreamingContext)`

`NameValueCollection(SerializationInfo, StreamingContext)`

Initializes a new instance of the NameValueCollection class that is serializable and uses the specified SerializationInfo and StreamingContext.

`NameValueCollection(Int32, IHashCodeProvider, IComparer)`

`NameValueCollection(Int32, IHashCodeProvider, IComparer)`

Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity and uses the specified hash code provider and the specified comparer.

Properties

`AllKeys`

`AllKeys`

Gets all the keys in the NameValueCollection.

`Item[Int32]`

`Item[Int32]`

Gets the entry at the specified index of the NameValueCollection.

`Item[String]`

`Item[String]`

Gets or sets the entry with the specified key in the NameValueCollection.

Methods

`Add(NameValueCollection)`

`Add(NameValueCollection)`

Copies the entries in the specified NameValueCollection to the current NameValueCollection.

`Add(String, String)`

`Add(String, String)`

Adds an entry with the specified name and value to the NameValueCollection.

`Clear()`

`Clear()`

Invalidates the cached arrays and removes all entries from the NameValueCollection.

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

Copies the entire NameValueCollection to a compatible one-dimensional Array, starting at the specified index of the target array.

`Get(Int32)`

`Get(Int32)`

Gets the values at the specified index of the NameValueCollection combined into one comma-separated list.

`Get(String)`

`Get(String)`

Gets the values associated with the specified key from the NameValueCollection combined into one comma-separated list.

`GetKey(Int32)`

`GetKey(Int32)`

Gets the key at the specified index of the NameValueCollection.

`GetValues(Int32)`

`GetValues(Int32)`

Gets the values at the specified index of the NameValueCollection.

`GetValues(String)`

`GetValues(String)`

Gets the values associated with the specified key from the NameValueCollection.

`HasKeys()`

`HasKeys()`

Gets a value indicating whether the NameValueCollection contains keys that are not `null`.

`InvalidateCachedArrays()`

`InvalidateCachedArrays()`

Resets the cached arrays of the collection to `null`.

`Remove(String)`

`Remove(String)`

Removes the entries with the specified key from the NameObjectCollectionBase instance.

`Set(String, String)`

`Set(String, String)`

Sets the value of an entry in the NameValueCollection.

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 NameValueCollection, but derived classes can create their own synchronized versions of the NameValueCollection using the SyncRoot property of the NameObjectCollectionBase class.

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

NameObjectCollectionBase NameObjectCollectionBase
NameObjectCollectionBase NameObjectCollectionBase

NameValueCollection.Add NameValueCollection.Add

In this Article

Overloads

Add(NameValueCollection) Add(NameValueCollection)	Copies the entries in the specified NameValueCollection to the current NameValueCollection.
Add(String, String) Add(String, String)	Adds an entry with the specified name and value to the NameValueCollection.

Add(NameValueCollection) Add(NameValueCollection)

Copies the entries in the specified NameValueCollection to the current NameValueCollection.

```
public void Add (System.Collections.Specialized.NameValueCollection c);
```

```
member this.Add : System.Collections.Specialized.NameValueCollection -> unit
```

Parameters

c NameValueCollection NameValueCollection

The NameValueCollection to copy to the current NameValueCollection.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

ArgumentNullException ArgumentNullException

`c` is `null`.

Remarks

If a key in `c` already exists in the target NameValueCollection instance, the associated value in `c` is added to the existing comma-separated list of values associated with the same key in the target NameValueCollection instance.

If Count already equals the capacity, the capacity of the NameValueCollection is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If Count is less than the capacity, 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 Also Set(String, String)Set(String, String)

Add(String, String) Add(String, String)

Adds an entry with the specified name and value to the NameValueCollection.

```
public virtual void Add (string name, string value);
```

```
abstract member Add : string * string -> unit
override this.Add : string * string -> unit
```

Parameters

name String String

The String key of the entry to add. The key can be `null`.

value String String

The String value of the entry to add. The value can be `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Remarks

If the specified key already exists in the target NameValueCollection instance, the specified value is added to the existing comma-separated list of values in the form `"value1,value2,value3"`. The values are associated with the same key in the target NameValueCollection instance.

If Count already equals the capacity, the capacity of the NameValueCollection is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If Count is less than the capacity, 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.

NameValueCollection.AllKeys NameValueCollection.AllKeys

In this Article

Gets all the keys in the NameValueCollection.

```
public virtual string[] AllKeys { get; }
```

```
member this.AllKeys : string[]
```

Returns
String[]

A String array that contains all the keys of the NameValueCollection.

Remarks

If the collection is empty, this method returns an empty String array, not `null`.

The arrays returned by AllKeys are cached for better performance and are automatically refreshed when the collection changes. A derived class can invalidate the cached version by calling InvalidateCachedArrays, thereby forcing the arrays to be recreated.

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

See Also	GetKey(Int32)GetKey(Int32) InvalidateCachedArrays()InvalidateCachedArrays()

NameValueCollection.Clear NameValueCollection.Clear

In this Article

Invalidates the cached arrays and removes all entries from the NameValueCollection.

```
public virtual void Clear ();
```

```
abstract member Clear : unit -> unit
override this.Clear : unit -> unit
```

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Remarks

This method is an O(1) operation.

See Also

InvalidateCachedArrays()InvalidateCachedArrays()

NameValueCollection.CopyTo NameValueCollection.CopyTo

In this Article

Copies the entire NameValueCollection to a compatible one-dimensional Array, starting at the specified index of the target array.

```
public void CopyTo (Array dest, int index);
```

```
member this.CopyTo : Array * int -> unit
```

Parameters

dest Array Array

The one-dimensional Array that is the destination of the elements copied from NameValueCollection. The Array must have zero-based indexing.

index Int32 Int32

The zero-based index in `dest` at which copying begins.

Exceptions

ArgumentNullException ArgumentNullException

`dest` is `null`.

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than zero.

ArgumentException ArgumentException

`dest` is multidimensional.

-or-

The number of elements in the source NameValueCollection is greater than the available space from `index` to the end of the destination `dest`.

InvalidCastException InvalidCastException

The type of the source NameValueCollection cannot be cast automatically to the type of the destination `dest`.

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.

NameValueCollection.Get NameValueCollection.Get

In this Article

Overloads

Get(Int32) Get(Int32)	Gets the values at the specified index of the NameValueCollection combined into one comma-separated list.
Get(String) Get(String)	Gets the values associated with the specified key from the NameValueCollection combined into one comma-separated list.

Get(Int32) Get(Int32)

Gets the values at the specified index of the NameValueCollection combined into one comma-separated list.

```
public virtual string Get (int index);
```

```
abstract member Get : int -> string
override this.Get : int -> string
```

Parameters

index Int32 Int32

The zero-based index of the entry that contains the values to get from the collection.

Returns

String String

A String that contains a comma-separated list of the values at the specified index of the NameValueCollection, if found; otherwise, `null`.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Remarks

This method is an O(`n`) operation, where `n` is the number of values at the specified index.

See Also

GetValues(String)GetValues(String)
AllKeysAllKeys
Performing Culture-Insensitive String Operations

Get(String) Get(String)

Gets the values associated with the specified key from the NameValueCollection combined into one comma-separated list.

```
public virtual string Get (string name);
```

```
abstract member Get : string -> string
override this.Get : string -> string
```

Parameters

name String String

The String key of the entry that contains the values to get. The key can be `null`.

Returns

String String

A String that contains a comma-separated list of the values associated with the specified key from the NameValueCollection, if found; otherwise, `null`.

Remarks

Caution

This method returns `null` in the following cases: 1) if the specified key is not found; and 2) if the specified key is found and its associated value is `null`. This method does not distinguish between the two cases.

This method is an O(`n`) operation, where `n` is the number of values associated with the specified key.

See Also

GetValues(String)GetValues(String)
AllKeysAllKeys
Performing Culture-Insensitive String Operations

NameValueCollection.GetKey NameValueCollection.GetKey

In this Article

Gets the key at the specified index of the NameValueCollection.

```
public virtual string GetKey (int index);
```

```
abstract member GetKey : int -> string
override this.GetKey : int -> string
```

Parameters

index Int32 Int32

The zero-based index of the key to get from the collection.

Returns

String String

A String that contains the key at the specified index of the NameValueCollection, if found; otherwise, `null`.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Remarks

This method is an O(1) operation.

See Also AllKeysAllKeys

NameValueCollection.GetValues NameValueCollection.GetValues

In this Article

Overloads

GetValues(Int32) GetValues(Int32)	Gets the values at the specified index of the NameValueCollection.
GetValues(String) GetValues(String)	Gets the values associated with the specified key from the NameValueCollection.

GetValues(Int32) GetValues(Int32)

Gets the values at the specified index of the NameValueCollection.

```
public virtual string[] GetValues (int index);
```

```
abstract member GetValues : int -> string[]
override this.GetValues : int -> string[]
```

Parameters

index Int32 Int32

The zero-based index of the entry that contains the values to get from the collection.

Returns

String[]

A String array that contains the values at the specified index of the NameValueCollection, if found; otherwise, `null`.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Remarks

This method is an O(`n`) operation, where `n` is the number of values at the specified index.

See Also

Get(String)Get(String)
AllKeysAllKeys
Performing Culture-Insensitive String Operations

GetValues(String) GetValues(String)

Gets the values associated with the specified key from the NameValueCollection.

```
public virtual string[] GetValues (string name);
```

```
abstract member GetValues : string -> string[]
override this.GetValues : string -> string[]
```

Parameters

name String String

The String key of the entry that contains the values to get. The key can be `null`.

Returns

String[]

A String array that contains the values associated with the specified key from the NameValueCollection, if found; otherwise, `null`.

Remarks

Caution

This method returns `null` in the following cases: 1) if the specified key is not found; and 2) if the specified key is found and its associated value is `null`. This method does not distinguish between the two cases.

This method is an O(`n`) operation, where `n` is the number of values associated with the specified key.

See Also

Get(String)Get(String)
AllKeysAllKeys
Performing Culture-Insensitive String Operations

NameValueCollection.HasKeys NameValueCollection.HasKeys

In this Article

Gets a value indicating whether the NameValueCollection contains keys that are not `null`.

```
public bool HasKeys ();
```

```
member this.HasKeys : unit -> bool
```

Returns

Boolean Boolean

`true` if the NameValueCollection contains keys that are not `null`; otherwise, `false`.

Remarks

This method is an O(1) operation.

NameValueCollection.InvalidateCachedArrays NameValueCollection.InvalidateCachedArrays

In this Article

Resets the cached arrays of the collection to `null`.

```
protected void InvalidateCachedArrays ();
```

```
member this.InvalidateCachedArrays : unit -> unit
```

Remarks

The arrays returned by AllKeys are cached for better performance and are automatically refreshed when the collection changes. A derived class can invalidate the cached version by calling InvalidateCachedArrays, thereby forcing the arrays to be recreated.

This method is an O(1) operation.

See Also AllKeysAllKeys

NameValueCollection.Item[String] NameValueCollection.Item[String]

In this Article

Overloads

Item[Int32] Item[Int32]	Gets the entry at the specified index of the NameValueCollection.
Item[String] Item[String]	Gets or sets the entry with the specified key in the NameValueCollection.

Item[Int32] Item[Int32]

Gets the entry at the specified index of the NameValueCollection.

```
public string this[int index] { get; }
```

```
member this.Item(int) : string
```

Parameters

index Int32 Int32

The zero-based index of the entry to locate in the collection.

Returns
String String

A String that contains the comma-separated list of values at the specified index of the collection.

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is outside the valid range of indexes for the collection.

Remarks

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[index]`.

This property cannot be set. To set the value at a specified index, use `Item[GetKey(index)]`.

The C# language uses the keyword to define the indexers instead of implementing the Item[String] property. Visual Basic implements Item[String] as a default property, which provides the same indexing functionality.

Retrieving the values at the specified index is an O(`n`) operation, where `n` is the number of values.

See Also Performing Culture-Insensitive String Operations

Item[String] Item[String]

Gets or sets the entry with the specified key in the NameValueCollection.

Gets or sets the entry with the specified key in the NameValueCollection.

```
public string this[string name] { get; set; }
```

```
member this.Item(string) : string with get, set
```

Parameters

name String String

The String key of the entry to locate. The key can be `null`.

Returns

String String

A String that contains the comma-separated list of values associated with the specified key, if found; otherwise, `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only and the operation attempts to modify the collection.

Remarks

This property provides the ability to access a specific element in the collection by using the following syntax: `myCollection[name]`.

If the specified key already exists in the collection, setting this property overwrites the existing list of values with the specified value. To append the new value to the existing list of values, use the Add method.

If the specified key does not exist in the collection, setting this property creates a new entry using the specified key and the specified value.

Caution

This property returns `null` in the following cases: 1) if the specified key is not found; and 2) if the specified key is found and its associated value is `null`. This property does not distinguish between the two cases.

The C# language uses the keyword to define the indexers instead of implementing the Item[String] property. Visual Basic implements Item[String] as a default property, which provides the same indexing functionality.

Retrieving or setting the values associated with the specified key is an O(`n`) operation, where `n` is the number of values.

See Also Performing Culture-Insensitive String Operations

NameValueCollection NameValueCollection

In this Article

Overloads

NameValueCollection()	Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity and uses the default case-insensitive hash code provider and the default case-insensitive comparer.
NameValueCollection(IEqualityComparer) NameValueCollection(IEqualityComparer)	Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity, and uses the specified IEqualityComparer object.
NameValueCollection(NameValueCollection) NameValueCollection(NameValueCollection)	Copies the entries from the specified NameValueCollection to a new NameValueCollection with the same initial capacity as the number of entries copied and using the same hash code provider and the same comparer as the source collection.
NameValueCollection(Int32) NameValueCollection(Int32)	Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity and uses the default case-insensitive hash code provider and the default case-insensitive comparer.
NameValueCollection(IHashCodeProvider, IComparer) NameValueCollection(IHashCodeProvider, IComparer)	Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity and uses the specified hash code provider and the specified comparer.
NameValueCollection(Int32, IEqualityComparer) NameValueCollection(Int32, IEqualityComparer)	Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer object.
NameValueCollection(Int32, NameValueCollection) NameValueCollection(Int32, NameValueCollection)	Copies the entries from the specified NameValueCollection to a new NameValueCollection with the specified initial capacity or the same initial capacity as the number of entries copied, whichever is greater, and using the default case-insensitive hash code provider and the default case-insensitive comparer.
NameValueCollection(SerializationInfo, StreamingContext) NameValueCollection(SerializationInfo, StreamingContext)	Initializes a new instance of the NameValueCollection class that is serializable and uses the specified SerializationInfo and StreamingContext.
NameValueCollection(Int32, IHashCodeProvider, IComparer) NameValueCollection(Int32, IHashCodeProvider, IComparer)	Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity and uses the specified hash code provider and the specified comparer.

NameValueCollection()

Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity and uses the default case-insensitive hash code provider and the default case-insensitive comparer.

```
public NameValueCollection ();
```

Remarks

The capacity of a NameValueCollection is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The hash code provider dispenses hash codes for keys in the NameValueCollection. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

This constructor is an O(1) operation.

See Also: Performing Culture-Insensitive String Operations

NameValueCollection(IEqualityComparer) NameValueCollection(IEqualityComparer)

Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity, and uses the specified IEqualityComparer object.

```
public NameValueCollection (System.Collections.IEqualityComparer equalityComparer);
```

```
new System.Collections.Specialized.NameValueCollection : System.Collections.IEqualityComparer ->
System.Collections.Specialized.NameValueCollection
```

Parameters

equalityComparer IEqualityComparer IEqualityComparer

The IEqualityComparer object to use to determine whether two keys are equal and to generate hash codes for the keys in the collection.

Remarks

The capacity of a NameValueCollection object is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The IEqualityComparer object combines the comparer and the hash code provider. The hash code provider dispenses hash codes for keys in the NameValueCollection. The comparer determines whether two keys are equal.

This constructor is an O(1) operation.

See IEqualityComparerIEqualityComparer

Also Performing Culture-Insensitive String Operations

NameValueCollection(NameValueCollection) NameValueCollection(NameValueCollection)

Copies the entries from the specified NameValueCollection to a new NameValueCollection with the same initial capacity as the number of entries copied and using the same hash code provider and the same comparer as the source collection.

```
public NameValueCollection (System.Collections.Specialized.NameValueCollection col);
```

```
new System.Collections.Specialized.NameValueCollection :
System.Collections.Specialized.NameValueCollection ->
System.Collections.Specialized.NameValueCollection
```

Parameters

col NameValueCollection NameValueCollection

The NameValueCollection to copy to the new NameValueCollection instance.

Exceptions
ArgumentNullException ArgumentNullException

`col` is `null`.

Remarks

The capacity of a NameValueCollection is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The hash code provider dispenses hash codes for keys in the NameValueCollection. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

The elements of the new NameValueCollection are sorted in the same order as the source NameValueCollection.

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

See Also Performing Culture-Insensitive String Operations

NameValueCollection(Int32) NameValueCollection(Int32)

Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity and uses the default case-insensitive hash code provider and the default case-insensitive comparer.

```
public NameValueCollection (int capacity);
```

```
new System.Collections.Specialized.NameValueCollection : int ->
System.Collections.Specialized.NameValueCollection
```

Parameters

capacity Int32 Int32

The initial number of entries that the NameValueCollection can contain.

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

Remarks

The capacity of a NameValueCollection is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The hash code provider dispenses hash codes for keys in the NameValueCollection. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

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

See Also Performing Culture-Insensitive String Operations

NameValueCollection(IHashCodeProvider, IComparer) NameValueCollection(IHashCodeProvider, IComparer)

Initializes a new instance of the NameValueCollection class that is empty, has the default initial capacity and uses the specified hash code provider and the specified comparer.

```
[System.Obsolete("Please use NameValueCollection(IEqualityComparer) instead.")]
public NameValueCollection (System.Collections.IHashCodeProvider hashProvider,
System.Collections.IComparer comparer);
```

```
new System.Collections.Specialized.NameValueCollection : System.Collections.IHashCodeProvider *
System.Collections.IComparer -> System.Collections.Specialized.NameValueCollection
```

Parameters

hashProvider IHashCodeProvider IHashCodeProvider

The IHashCodeProvider that will supply the hash codes for all keys in the NameValueCollection.

comparer IComparer IComparer

The IComparer to use to determine whether two keys are equal.

Attributes ObsoleteAttribute

Remarks

The capacity of a NameValueCollection is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The hash code provider dispenses hash codes for keys in the NameValueCollection. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

This constructor is an O(1) operation.

See Also
IHashCodeProviderIHashCodeProvider
IComparerIComparer
Performing Culture-Insensitive String Operations

NameValueCollection(Int32, IEqualityComparer) NameValueCollection(Int32, IEqualityComparer)

Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer object.

```
public NameValueCollection (int capacity, System.Collections.IEqualityComparer equalityComparer);
```

```
new System.Collections.Specialized.NameValueCollection : int * System.Collections.IEqualityComparer
-> System.Collections.Specialized.NameValueCollection
```

Parameters

capacity
Int32 Int32

The initial number of entries that the NameValueCollection object can contain.

equalityComparer
IEqualityComparer IEqualityComparer

The IEqualityComparer object to use to determine whether two keys are equal and to generate hash codes for the keys in the collection.

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

Remarks

The capacity of a NameValueCollection object is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The IEqualityComparer object combines the comparer and the hash code provider. The hash code provider dispenses hash codes for keys in the NameValueCollection. The comparer determines whether two keys are equal.

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

See Also
IEqualityComparerIEqualityComparer
Performing Culture-Insensitive String Operations

NameValueCollection(Int32, NameValueCollection)

NameValueCollection(Int32, NameValueCollection)

Copies the entries from the specified NameValueCollection to a new NameValueCollection with the specified initial capacity or the same initial capacity as the number of entries copied, whichever is greater, and using the default case-insensitive hash code provider and the default case-insensitive comparer.

```
public NameValueCollection (int capacity, System.Collections.Specialized.NameValueCollection col);
```

```
new System.Collections.Specialized.NameValueCollection : int *
System.Collections.Specialized.NameValueCollection ->
System.Collections.Specialized.NameValueCollection
```

Parameters

capacity Int32 Int32

The initial number of entries that the NameValueCollection can contain.

col NameValueCollection NameValueCollection

The NameValueCollection to copy to the new NameValueCollection instance.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

ArgumentNullException ArgumentNullException

`col` is `null`.

Remarks

The capacity of a NameValueCollection is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The hash code provider dispenses hash codes for keys in the NameValueCollection. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

This constructor is an O(`n`) operation, where `n` is `capacity`. If the number of elements in `col` is greater than `capacity`, this constructor becomes an O(`n` + `m`) operation, where `n` is `capacity` and `m` is the number of elements in `col`.

See Also Performing Culture-Insensitive String Operations

NameValueCollection(SerializationInfo, StreamingContext) NameValueCollection(SerializationInfo, StreamingContext)

Initializes a new instance of the NameValueCollection class that is serializable and uses the specified SerializationInfo and StreamingContext.

```
protected NameValueCollection (System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);
```

```
new System.Collections.Specialized.NameValueCollection :
System.Runtime.Serialization.SerializationInfo * System.Runtime.Serialization.StreamingContext ->
System.Collections.Specialized.NameValueCollection
```

Parameters

info SerializationInfo SerializationInfo

A SerializationInfo object that contains the information required to serialize the new NameValueCollection instance.

context StreamingContext StreamingContext

A StreamingContext object that contains the source and destination of the serialized stream associated with the new NameValueCollection instance.

Remarks

This constructor is an O(1) operation.

See Also
SerializationInfoSerializationInfo
StreamingContextStreamingContext
Performing Culture-Insensitive String Operations

NameValueCollection(Int32, IHashCodeProvider, IComparer) NameValueCollection(Int32, IHashCodeProvider, IComparer)

Initializes a new instance of the NameValueCollection class that is empty, has the specified initial capacity and uses the specified hash code provider and the specified comparer.

```
[System.Obsolete("Please use NameValueCollection(Int32, IEqualityComparer) instead.")]
public NameValueCollection (int capacity, System.Collections.IHashCodeProvider hashProvider,
System.Collections.IComparer comparer);
```

```
new System.Collections.Specialized.NameValueCollection : int * System.Collections.IHashCodeProvider
* System.Collections.IComparer -> System.Collections.Specialized.NameValueCollection
```

Parameters

capacity Int32 Int32

The initial number of entries that the NameValueCollection can contain.

hashProvider IHashCodeProvider IHashCodeProvider

The IHashCodeProvider that will supply the hash codes for all keys in the NameValueCollection.

comparer IComparer IComparer

The IComparer to use to determine whether two keys are equal.

Attributes ObsoleteAttribute

Exceptions
ArgumentOutOfRangeException ArgumentOutOfRangeException

`capacity` is less than zero.

Remarks

The capacity of a NameValueCollection is the number of elements that the NameValueCollection can hold. As elements are added to a NameValueCollection, 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 NameValueCollection.

The hash code provider dispenses hash codes for keys in the NameValueCollection. The default hash code provider is the CaseInsensitiveHashCodeProvider.

The comparer determines whether two keys are equal. The default comparer is the CaseInsensitiveComparer.

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

See Also	Performing Culture-Insensitive String Operations

NameValueCollection.Remove NameValueCollection.Remove

In this Article

Removes the entries with the specified key from the NameObjectCollectionBase instance.

```
public virtual void Remove (string name);
```

```
abstract member Remove : string -> unit
override this.Remove : string -> unit
```

Parameters

name String String

The String key of the entry to remove. The key can be `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Remarks

If the NameValueCollection doesn't contain an element with the specified key, the NameValueCollection remains unchanged. No exception is thrown.

If you specify a `null` value to the `name` parameter, an entry with a `null` key is removed, if found.

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 Also

Performing Culture-Insensitive String Operations

NameValueCollection.Set NameValueCollection.Set

In this Article

Sets the value of an entry in the NameValueCollection.

```
public virtual void Set (string name, string value);
```

```
abstract member Set : string * string -> unit
override this.Set : string * string -> unit
```

Parameters

name String String

The String key of the entry to add the new value to. The key can be `null`.

value String String

The Object that represents the new value to add to the specified entry. The value can be `null`.

Exceptions

NotSupportedException NotSupportedException

The collection is read-only.

Remarks

If the specified key already exists in the collection, this method overwrites the existing list of values with the specified value. To append the new value to the existing list of values, use the Add method.

If the specified key does not exist in the collection, this method creates a new entry using the specified key and the specified value.

This method is an O(1) operation.

See Also

Add(NameValueCollection)Add(NameValueCollection)

Performing Culture-Insensitive String Operations

NotifyCollectionChangedAction NotifyCollectionChangedAction Enum

Describes the action that caused a CollectionChanged event.

Declaration

```
public enum NotifyCollectionChangedAction
```

```
type NotifyCollectionChangedAction =
```

Inheritance Hierarchy

`Object` `Object`
  `ValueType` `ValueType`
    `Enum` `Enum`

Fields

`Add` `Add`	An item was added to the collection.
`Move` `Move`	An item was moved within the collection.
`Remove` `Remove`	An item was removed from the collection.
`Replace` `Replace`	An item was replaced in the collection.
`Reset` `Reset`	The content of the collection was cleared.

See Also

NotifyCollectionChangedEventArgs NotifyCollectionChangedEventArgs Class

Provides data for the CollectionChanged event.

Declaration

```
public class NotifyCollectionChangedEventArgs : EventArgs
```

```
type NotifyCollectionChangedEventArgs = class
    inherit EventArgs
```

Inheritance Hierarchy

Object Object

EventArgs EventArgs

Constructors

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a Reset change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Replace change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item change or a Reset change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Replace change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList, Int32)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Replace change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32, Int32)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Move change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32, Int32)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Move change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object, Int32)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Replace change.

Properties

Action

Action

Gets the action that caused the event.

NewItems

NewItems

Gets the list of new items involved in the change.

NewStartingIndex

NewStartingIndex

Gets the index at which the change occurred.

OldItems

OldItems

Gets the list of items affected by a Replace, Remove, or Move action.

OldStartingIndex

OldStartingIndex

Gets the index at which a Move, Remove, or Replace action occurred.

NotifyCollectionChangedEventArgs.Action NotifyCollectionChangedEventArgs.Action

In this Article

Gets the action that caused the event.

```
public System.Collections.Specialized.NotifyCollectionChangedAction Action { get; }
```

```
member this.Action : System.Collections.Specialized.NotifyCollectionChangedAction
```

Returns

NotifyCollectionChangedAction NotifyCollectionChangedAction

A NotifyCollectionChangedAction value that describes the action that caused the event.

NotifyCollectionChangedEventArgs.NewItems NotifyCollectionChangedEventArgs.NewItems

In this Article

Gets the list of new items involved in the change.

```
public System.Collections.IList NewItems { get; }
```

```
member this.NewItems : System.Collections.IList
```

Returns

IList IList

The list of new items involved in the change.

NotifyCollectionChangedEventArgs.NewStartingIndex NotifyCollectionChangedEventArgs.NewStartingIndex

In this Article

Gets the index at which the change occurred.

```
public int NewStartingIndex { get; }
```

```
member this.NewStartingIndex : int
```

Returns

Int32 Int32

The zero-based index at which the change occurred.

NotifyCollectionChangedEventArgs NotifyCollectionChangedEventArgs

In this Article

Overloads

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a Reset change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Replace change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item change or a Reset change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Replace change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList, Int32)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Replace change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32, Int32)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Move change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32, Int32)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Move change.
NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object, Int32)	Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Replace change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a Reset change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This must be set to Reset.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, System.Collections.IList
changedItems);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * System.Collections.IList ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can be set to Reset, Add, or Remove.

changedItems IList IList

The items that are affected by the change.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, object changedItem);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * obj ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can be set to Reset, Add, or Remove.

changedItem Object Object

The item that is affected by the change.

Exceptions

ArgumentException ArgumentException

If `action` is not Reset, Add, or Remove, or if `action` is Reset and `changedItem` is not null.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Replace change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, System.Collections.IList
newItems, System.Collections.IList oldItems);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * System.Collections.IList *
System.Collections.IList -> System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can only be set to Replace.

newItems IList IList

The new items that are replacing the original items.

oldItems IList IList

The original items that are replaced.

Exceptions
ArgumentException ArgumentException

If `action` is not Replace.

ArgumentNullException ArgumentNullException

If `oldItems` or `newItems` is null.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item change or a Reset change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, System.Collections.IList
changedItems, int startingIndex);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * System.Collections.IList * int ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can be set to Reset, Add, or Remove.

changedItems IList IList

The items affected by the change.

startingIndex Int32 Int32

The index where the change occurred.

Exceptions
ArgumentException ArgumentException

If `action` is not Reset, Add, or Remove, if `action` is Reset and either `changedItems` is not null or `startingIndex` is not -1, or if action is Add or Remove and `startingIndex` is less than -1.

ArgumentNullException ArgumentNullException

If `action` is Add or Remove and `changedItems` is null.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, object changedItem, int
index);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * obj * int ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can be set to Reset, Add, or Remove.

changedItem Object Object

The item that is affected by the change.

index Int32 Int32

The index where the change occurred.

Exceptions
ArgumentException ArgumentException

If `action` is not Reset, Add, or Remove, or if `action` is Reset and either `changedItems` is not null or `index` is not -1.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Replace change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, object newItem, object
oldItem);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * obj * obj ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can only be set to Replace.

newItem Object Object

The new item that is replacing the original item.

oldItem Object Object

The original item that is replaced.

Exceptions
ArgumentException ArgumentException

If `action` is not Replace.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, IList, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Replace change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, System.Collections.IList
newItems, System.Collections.IList oldItems, int startingIndex);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * System.Collections.IList *
System.Collections.IList * int -> System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can only be set to Replace.

newItems IList IList

The new items that are replacing the original items.

oldItems IList IList

The original items that are replaced.

startingIndex Int32 Int32

The index of the first item of the items that are being replaced.

Exceptions
ArgumentException ArgumentException

If `action` is not Replace.

ArgumentNullException ArgumentNullException

If `oldItems` or `newItems` is null.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, IList, Int32, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a multi-item Move change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, System.Collections.IList
changedItems, int index, int oldIndex);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * System.Collections.IList * int * int
-> System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can only be set to Move.

changedItems IList IList

The items affected by the change.

index Int32 Int32

The new index for the changed items.

oldIndex Int32 Int32

The old index for the changed items.

Exceptions
ArgumentException ArgumentException

If `action` is not Move or `index` is less than 0.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32, Int32) NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Int32, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Move change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, object changedItem, int index,
int oldIndex);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * obj * int * int ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can only be set to Move.

changedItem Object Object

The item affected by the change.

index Int32 Int32

The new index for the changed item.

oldIndex Int32 Int32

The old index for the changed item.

Exceptions
ArgumentException ArgumentException

If `action` is not Move or `index` is less than 0.

NotifyCollectionChangedEventArgs(NotifyCollectionChangedActio

n, Object, Object, Int32)

NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction, Object, Object, Int32)

Initializes a new instance of the NotifyCollectionChangedEventArgs class that describes a one-item Replace change.

```
public NotifyCollectionChangedEventArgs
(System.Collections.Specialized.NotifyCollectionChangedAction action, object newItem, object
oldItem, int index);
```

```
new System.Collections.Specialized.NotifyCollectionChangedEventArgs :
System.Collections.Specialized.NotifyCollectionChangedAction * obj * obj * int ->
System.Collections.Specialized.NotifyCollectionChangedEventArgs
```

Parameters

action NotifyCollectionChangedAction NotifyCollectionChangedAction

The action that caused the event. This can be set to Replace.

newItem Object Object

The new item that is replacing the original item.

oldItem Object Object

The original item that is replaced.

index Int32 Int32

The index of the item being replaced.

Exceptions

ArgumentException ArgumentException

If `action` is not Replace.

NotifyCollectionChangedEventArgs.OldItems NotifyCollectionChangedEventArgs.OldItems

In this Article

Gets the list of items affected by a Replace, Remove, or Move action.

```
public System.Collections.IList OldItems { get; }
```

```
member this.OldItems : System.Collections.IList
```

Returns

IList IList

The list of items affected by a Replace, Remove, or Move action.

NotifyCollectionChangedEventArgs.OldStartingIndex NotifyCollectionChangedEventArgs.OldStartingIndex

In this Article

Gets the index at which a Move, Remove, or Replace action occurred.

```
public int OldStartingIndex { get; }
```

```
member this.OldStartingIndex : int
```

Returns

Int32 Int32

The zero-based index at which a Move, Remove, or Replace action occurred.

NotifyCollectionChangedEventHandler NotifyCollectionChangedEventHandler Delegate

Represents the method that handles the CollectionChanged event.

Declaration

```
public delegate void NotifyCollectionChangedEventHandler(object sender,
NotifyCollectionChangedEventArgs e);
```

```
type NotifyCollectionChangedEventHandler = delegate of obj * NotifyCollectionChangedEventArgs ->
unit
```

Inheritance Hierarchy

`Object` `Object`

`Delegate` `Delegate`

OrderedDictionary OrderedDictionary Class

Represents a collection of key/value pairs that are accessible by the key or index.

Declaration

```
[Serializable]
public class OrderedDictionary : System.Collections.IDictionary,
System.Collections.Specialized.IOrderedDictionary,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

```
type OrderedDictionary = class
    interface IOrderedDictionary
    interface ISerializable
    interface IDeserializationCallback
    interface IDictionary
    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.

The elements of an OrderedDictionary are not sorted by the key, unlike the elements of a SortedDictionary<TKey,TValue> class. You can access elements either by the key or by the index.

The `foreach` statement of the C# language (`For Each` in Visual Basic) returns objects that are of the type of each element in the collection. Since each element of the OrderedDictionary collection 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 following code shows C#, Visual Basic and C++ syntax.

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

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

Constructors

`OrderedDictionary()`

`OrderedDictionary()`

Initializes a new instance of the OrderedDictionary class.

`OrderedDictionary(IEqualityComparer)`

`OrderedDictionary(IEqualityComparer)`

Initializes a new instance of the OrderedDictionary class using the specified comparer.

`OrderedDictionary(Int32)`

`OrderedDictionary(Int32)`

Initializes a new instance of the OrderedDictionary class using the specified initial capacity.

`OrderedDictionary(Int32, IEqualityComparer)`

`OrderedDictionary(Int32, IEqualityComparer)`

Initializes a new instance of the OrderedDictionary class using the specified initial capacity and comparer.

`OrderedDictionary(SerializationInfo, StreamingContext)`

`OrderedDictionary(SerializationInfo, StreamingContext)`

Initializes a new instance of the OrderedDictionary class that is serializable using the specified SerializationInfo and StreamingContext objects.

Properties

`Count`

`Count`

Gets the number of key/values pairs contained in the OrderedDictionary collection.

`IsReadOnly`

`IsReadOnly`

Gets a value indicating whether the OrderedDictionary collection is read-only.

`Item[Int32]`

`Item[Int32]`

Gets or sets the value at the specified index.

`Item[Object]`

`Item[Object]`

Gets or sets the value with the specified key.

`Keys`

`Keys`

Gets an ICollection object containing the keys in the OrderedDictionary collection.

`Values`

`Values`

Gets an ICollection object containing the values in the OrderedDictionary collection.

Methods

`Add(Object, Object)`

`Add(Object, Object)`

Adds an entry with the specified key and value into the OrderedDictionary collection with the lowest available index.

`AsReadOnly()`

`AsReadOnly()`

Returns a read-only copy of the current OrderedDictionary collection.

`Clear()`

`Clear()`

Removes all elements from the OrderedDictionary collection.

`Contains(Object)`

`Contains(Object)`

Determines whether the OrderedDictionary collection contains a specific key.

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

Copies the OrderedDictionary elements to a one-dimensional Array object at the specified index.

`GetEnumerator()`

`GetEnumerator()`

Returns an IDictionaryEnumerator object that iterates through the OrderedDictionary collection.

`GetObjectData(SerializationInfo, StreamingContext)`

`GetObjectData(SerializationInfo, StreamingContext)`

Implements the ISerializable interface and returns the data needed to serialize the OrderedDictionary collection.

`Insert(Int32, Object, Object)`

`Insert(Int32, Object, Object)`

Inserts a new entry into the OrderedDictionary collection with the specified key and value at the specified index.

`OnDeserialization(Object)`

`OnDeserialization(Object)`

Implements the ISerializable interface and is called back by the deserialization event when deserialization is complete.

`Remove(Object)`

`Remove(Object)`

Removes the entry with the specified key from the OrderedDictionary collection.

`RemoveAt(Int32)`

`RemoveAt(Int32)`

Removes the entry at the specified index from the OrderedDictionary collection.

`ICollection.IsSynchronized`

`ICollection.IsSynchronized`

Gets a value indicating whether access to the OrderedDictionary object is synchronized (thread-safe).

`ICollection.SyncRoot`

`ICollection.SyncRoot`

Gets an object that can be used to synchronize access to the OrderedDictionary object.

`IDictionary.IsFixedSize`

`IDictionary.IsFixedSize`

Gets a value indicating whether the OrderedDictionary has a fixed size.

`IEnumerable.GetEnumerator()`

`IEnumerable.GetEnumerator()`

Returns an IDictionaryEnumerator object that iterates through the OrderedDictionary collection.

`IDeserializationCallback.OnDeserialization(Object)`

`IDeserializationCallback.OnDeserialization(Object)`

Implements the ISerializable interface and is called back by the deserialization event when deserialization is complete.

OrderedDictionary.Add OrderedDictionary.Add

In this Article

Adds an entry with the specified key and value into the OrderedDictionary collection with the lowest available index.

```
public 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 entry to add.

value Object Object

The value of the entry to add. This value can be `null`.

Exceptions

NotSupportedException NotSupportedException

The OrderedDictionary collection is read-only.

ArgumentException ArgumentException

An element with the same key already exists in the OrderedDictionary collection.

Examples

The following code example demonstrates the creation and population of an OrderedDictionary collection. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Creates and initializes a OrderedDictionary.
OrderedDictionary myOrderedDictionary = new OrderedDictionary();
myOrderedDictionary.Add("testKey1", "testValue1");
myOrderedDictionary.Add("testKey2", "testValue2");
myOrderedDictionary.Add("keyToDelete", "valueToDelete");
myOrderedDictionary.Add("testKey3", "testValue3");

ICollection keyCollection = myOrderedDictionary.Keys;
ICollection valueCollection = myOrderedDictionary.Values;

// Display the contents using the key and value collections
DisplayContents(keyCollection, valueCollection, myOrderedDictionary.Count);
```

Remarks

A key cannot be `null`, but a value can be.

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 OrderedDictionary collection; however, if the specified key already exists in the OrderedDictionary, setting the Item[Object] property overwrites the old value. In contrast, the Add method does not modify existing elements but instead throws ArgumentException.

OrderedDictionary.AsReadOnly OrderedDictionary.AsReadOnly

In this Article

Returns a read-only copy of the current OrderedDictionary collection.

```
public System.Collections.Specialized.OrderedDictionary AsReadOnly ();
```

```
member this.AsReadOnly : unit -> System.Collections.Specialized.OrderedDictionary
```

Returns
OrderedDictionary OrderedDictionary

A read-only copy of the current OrderedDictionary collection.

Remarks

The AsReadOnly method creates a read-only wrapper around the current OrderedDictionary collection. Changes made to the OrderedDictionary collection are reflected in the read-only copy.

OrderedDictionary.Clear OrderedDictionary.Clear

In this Article

Removes all elements from the OrderedDictionary collection.

```
public void Clear ();
```

```
abstract member Clear : unit -> unit
override this.Clear : unit -> unit
```

Exceptions

NotSupportedException NotSupportedException

The OrderedDictionary collection is read-only.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the Clear method is used to empty the OrderedDictionary, and then the OrderedDictionary is repopulated. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Clear the OrderedDictionary and add new values
myOrderedDictionary.Clear();
myOrderedDictionary.Add("newKey1", "newValue1");
myOrderedDictionary.Add("newKey2", "newValue2");
myOrderedDictionary.Add("newKey3", "newValue3");

// Display the contents of the "new" Dictionary using an enumerator
IDictionaryEnumerator myEnumerator =
    myOrderedDictionary.GetEnumerator();

Console.WriteLine(
    "{0}Displaying the entries of a \"new\" OrderedDictionary.",
    Environment.NewLine);

DisplayEnumerator(myEnumerator);
```

Remarks

After calling the Clear method, the Count property is set to zero and references to other objects from elements of the collection are also released. The capacity is not changed as a result of calling this method.

OrderedDictionary.Contains OrderedDictionary.Contains

In this Article

Determines whether the OrderedDictionary collection contains a specific key.

```
public bool Contains (object key);
```

```
abstract member Contains : obj -> bool
override this.Contains : obj -> bool
```

Parameters

key Object Object

The key to locate in the OrderedDictionary collection.

Returns

Boolean Boolean

`true` if the OrderedDictionary collection contains an element with the specified key; otherwise, `false`.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the Contains method is used to determine if an entry exists before attempting to remove it. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Modifying the OrderedDictionary
if (!myOrderedDictionary.IsReadOnly)
{
    // Insert a new key to the beginning of the OrderedDictionary
    myOrderedDictionary.Insert(0, "insertedKey1", "insertedValue1");

    // Modify the value of the entry with the key "testKey2"
    myOrderedDictionary["testKey2"] = "modifiedValue";

    // Remove the last entry from the OrderedDictionary: "testKey3"
    myOrderedDictionary.RemoveAt(myOrderedDictionary.Count - 1);

    // Remove the "keyToDelete" entry, if it exists
    if (myOrderedDictionary.Contains("keyToDelete"))
    {
        myOrderedDictionary.Remove("keyToDelete");
    }
}
```

Remarks

Using the Item[Object] property can return a null value if the key does not exist or if the key is `null`. Use the Contains method to determine if a specific key exists in the OrderedDictionary collection.

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.

OrderedDictionary.CopyTo OrderedDictionary.CopyTo

In this Article

Copies the OrderedDictionary elements to a one-dimensional Array object 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 object that is the destination of the DictionaryEntry objects copied from OrderedDictionary collection. The Array must have zero-based indexing.

index Int32 Int32

The zero-based index in `array` at which copying begins.

Remarks

The CopyTo method is not guaranteed to preserve the order of the elements in the OrderedDictionary collection.

OrderedDictionary.Count OrderedDictionary.Count

In this Article

Gets the number of key/values pairs contained in the OrderedDictionary collection.

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

```
member this.Count : int
```

Returns

Int32 Int32

The number of key/value pairs contained in the OrderedDictionary collection.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the Count property is used to remove the last item in the OrderedDictionary. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Modifying the OrderedDictionary
if (!myOrderedDictionary.IsReadOnly)
{
    // Insert a new key to the beginning of the OrderedDictionary
    myOrderedDictionary.Insert(0, "insertedKey1", "insertedValue1");

    // Modify the value of the entry with the key "testKey2"
    myOrderedDictionary["testKey2"] = "modifiedValue";

    // Remove the last entry from the OrderedDictionary: "testKey3"
    myOrderedDictionary.RemoveAt(myOrderedDictionary.Count - 1);

    // Remove the "keyToDelete" entry, if it exists
    if (myOrderedDictionary.Contains("keyToDelete"))
    {
        myOrderedDictionary.Remove("keyToDelete");
    }
}
```

OrderedDictionary.GetEnumerator OrderedDictionary.GetEnumerator

In this Article

Returns an IDictionaryEnumerator object that iterates through the OrderedDictionary collection.

```
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 OrderedDictionary collection.

Examples

The following code example demonstrates the use of the GetEnumerator method to display the contents of the OrderedDictionary collection to the console. In this example, the GetEnumerator method is used to obtain an IDictionaryEnumerator object that is passed to a method that displays the contents. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Clear the OrderedDictionary and add new values
myOrderedDictionary.Clear();
myOrderedDictionary.Add("newKey1", "newValue1");
myOrderedDictionary.Add("newKey2", "newValue2");
myOrderedDictionary.Add("newKey3", "newValue3");

// Display the contents of the "new" Dictionary using an enumerator
IDictionaryEnumerator myEnumerator =
    myOrderedDictionary.GetEnumerator();

Console.WriteLine(
    "{0}Displaying the entries of a \"new\" OrderedDictionary.",
    Environment.NewLine);

DisplayEnumerator(myEnumerator);
```

```
// Displays the contents of the OrderedDictionary using its enumerator
public static void DisplayEnumerator(IDictionaryEnumerator myEnumerator)
{
    Console.WriteLine("   KEY                       VALUE");
    while (myEnumerator.MoveNext())
    {
        Console.WriteLine("   {0,-25} {1}",
            myEnumerator.Key, myEnumerator.Value);
    }
}
```

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.

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.

OrderedDictionary.GetObjectData OrderedDictionary.GetObjectData

In this Article

Implements the ISerializable interface and returns the data needed to serialize the OrderedDictionary collection.

```
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 OrderedDictionary collection.

context StreamingContext StreamingContext

A StreamingContext object containing the source and destination of the serialized stream associated with the OrderedDictionary.

Exceptions

ArgumentNullException ArgumentNullException

`info` is `null`.

OrderedDictionary.ICollection.IsSynchronized

In this Article

Gets a value indicating whether access to the OrderedDictionary object is synchronized (thread-safe).

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

Returns

Boolean

This method always returns `false`.

OrderedDictionary.ICollection.SyncRoot

In this Article

Gets an object that can be used to synchronize access to the OrderedDictionary object.

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

Returns

Object

An object that can be used to synchronize access to the OrderedDictionary object.

OrderedDictionary.IDeserializationCallback.OnDeserialization

In this Article

Implements the ISerializable interface and is called back by the deserialization event when deserialization is complete.

```
void IDeserializationCallback.OnDeserialization (object sender);
```

Parameters

sender Object

The source of the deserialization event.

OrderedDictionary.IDictionary.IsFixedSize

In this Article

Gets a value indicating whether the OrderedDictionary has a fixed size.

```
bool System.Collections.IDictionary.IsFixedSize { get; }
```

Returns

Boolean

`true` if the OrderedDictionary has a fixed size; otherwise, `false`. The default is `false`.

OrderedDictionary.IEnumerable.GetEnumerator

In this Article

Returns an IDictionaryEnumerator object that iterates through the OrderedDictionary collection.

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

Returns

IEnumerator

An IDictionaryEnumerator object for the OrderedDictionary collection.

OrderedDictionary.Insert OrderedDictionary.Insert

In this Article

Inserts a new entry into the OrderedDictionary collection with the specified key and value at the specified index.

```
public void Insert (int index, object key, object value);
```

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

Parameters

index Int32 Int32

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

key Object Object

The key of the entry to add.

value Object Object

The value of the entry to add. The value can be `null`.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is out of range.

NotSupportedException NotSupportedException

This collection is read-only.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the Insert method is used to add a new entry to the beginning of the OrderedDictionary, moving the rest of the entries down. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Modifying the OrderedDictionary
if (!myOrderedDictionary.IsReadOnly)
{
    // Insert a new key to the beginning of the OrderedDictionary
    myOrderedDictionary.Insert(0, "insertedKey1", "insertedValue1");

    // Modify the value of the entry with the key "testKey2"
    myOrderedDictionary["testKey2"] = "modifiedValue";

    // Remove the last entry from the OrderedDictionary: "testKey3"
    myOrderedDictionary.RemoveAt(myOrderedDictionary.Count - 1);

    // Remove the "keyToDelete" entry, if it exists
    if (myOrderedDictionary.Contains("keyToDelete"))
    {
        myOrderedDictionary.Remove("keyToDelete");
    }
}
```

Remarks

If the `index` parameter is equal to the number of entries in the OrderedDictionary collection, the `key` and `value` parameters are appended to the end of the collection.

Entries that follow the insertion point move down to accommodate the new entry and the indexes of the moved entries are also updated.

OrderedDictionary.IsReadOnly OrderedDictionary.IsReadOnly

In this Article

Gets a value indicating whether the OrderedDictionary collection is read-only.

```
public bool IsReadOnly { get; }
```

```
member this.IsReadOnly : bool
```

Returns

Boolean Boolean

`true` if the OrderedDictionary collection is read-only; otherwise, `false`. The default is `false`.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the IsReadOnly property is used to determine whether the OrderedDictionary can be modified. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Modifying the OrderedDictionary
if (!myOrderedDictionary.IsReadOnly)
{
    // Insert a new key to the beginning of the OrderedDictionary
    myOrderedDictionary.Insert(0, "insertedKey1", "insertedValue1");

    // Modify the value of the entry with the key "testKey2"
    myOrderedDictionary["testKey2"] = "modifiedValue";

    // Remove the last entry from the OrderedDictionary: "testKey3"
    myOrderedDictionary.RemoveAt(myOrderedDictionary.Count - 1);

    // Remove the "keyToDelete" entry, if it exists
    if (myOrderedDictionary.Contains("keyToDelete"))
    {
        myOrderedDictionary.Remove("keyToDelete");
    }
}
```

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 modification of the collection; therefore, if changes are made to the underlying collection, the read-only collection reflects those changes.

OrderedDictionary.Item[Object] OrderedDictionary.Item[Object]

In this Article

Overloads

Item[Int32] Item[Int32]	Gets or sets the value at the specified index.
Item[Object] Item[Object]	Gets or sets the value with the specified key.

Item[Int32] Item[Int32]

Gets or sets the value 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 value to get or set.

Returns
Object Object

The value of the item at the specified index.

Exceptions
NotSupportedException NotSupportedException

The property is being set and the OrderedDictionary collection is read-only.

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than zero.

-or-

`index` is equal to or greater than Count.

Remarks

This property allows you 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[Object] property. Visual Basic implements Item[Object] as a default property, which provides the same indexing functionality.

Item[Object] Item[Object]

Gets or sets the value 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 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
NotSupportedException NotSupportedException

The property is being set and the OrderedDictionary collection is read-only.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the Item[Object] property is used to modify the dictionary entry with the key `"testKey2"`. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Modifying the OrderedDictionary
if (!myOrderedDictionary.IsReadOnly)
{
    // Insert a new key to the beginning of the OrderedDictionary
    myOrderedDictionary.Insert(0, "insertedKey1", "insertedValue1");

    // Modify the value of the entry with the key "testKey2"
    myOrderedDictionary["testKey2"] = "modifiedValue";

    // Remove the last entry from the OrderedDictionary: "testKey3"
    myOrderedDictionary.RemoveAt(myOrderedDictionary.Count - 1);

    // Remove the "keyToDelete" entry, if it exists
    if (myOrderedDictionary.Contains("keyToDelete"))
    {
        myOrderedDictionary.Remove("keyToDelete");
    }
}
```

Remarks

This property allows you 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 OrderedDictionary collection (for example, `myCollection["myNonexistentKey"] = myValue`). However, if the specified key already exists in the OrderedDictionary, 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 to determine if the key exists in the OrderedDictionary.

OrderedDictionary.Keys OrderedDictionary.Keys

In this Article

Gets an ICollection object containing the keys in the OrderedDictionary collection.

```
public System.Collections.ICollection Keys { get; }
```

```
member this.Keys : System.Collections.ICollection
```

Returns
ICollection ICollection

An ICollection object containing the keys in the OrderedDictionary collection.

Examples

The following code example demonstrates the creation and population of an OrderedDictionary collection, and then prints the contents to the console. In this example, the Keys and Values properties are passed to a method that displays the contents. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Creates and initializes a OrderedDictionary.
OrderedDictionary myOrderedDictionary = new OrderedDictionary();
myOrderedDictionary.Add("testKey1", "testValue1");
myOrderedDictionary.Add("testKey2", "testValue2");
myOrderedDictionary.Add("keyToDelete", "valueToDelete");
myOrderedDictionary.Add("testKey3", "testValue3");

ICollection keyCollection = myOrderedDictionary.Keys;
ICollection valueCollection = myOrderedDictionary.Values;

// Display the contents using the key and value collections
DisplayContents(keyCollection, valueCollection, myOrderedDictionary.Count);
```

```
// Displays the contents of the OrderedDictionary from its keys and values
public static void DisplayContents(
    ICollection keyCollection, ICollection valueCollection, int dictionarySize)
{
    String[] myKeys = new String[dictionarySize];
    String[] myValues = new String[dictionarySize];
    keyCollection.CopyTo(myKeys, 0);
    valueCollection.CopyTo(myValues, 0);

    // Displays the contents of the OrderedDictionary
    Console.WriteLine("   INDEX KEY                       VALUE");
    for (int i = 0; i < dictionarySize; i++)
    {
        Console.WriteLine("   {0,-5} {1,-25} {2}",
            i, myKeys[i], myValues[i]);
    }
    Console.WriteLine();
}
```

Remarks

The returned ICollection object is not a static copy; instead, the ICollection refers back to the keys in the original OrderedDictionary collection. Therefore, changes to the OrderedDictionary continue to be reflected in the ICollection.

OrderedDictionary.OnDeserialization OrderedDictionary.OnDeserialization

In this Article

Implements the ISerializable interface and is called back by the deserialization event when deserialization is complete.

```
protected virtual void OnDeserialization (object sender);
```

```
abstract member OnDeserialization : obj -> unit
override this.OnDeserialization : obj -> unit
```

Parameters

sender Object Object

The source of the deserialization event.

Exceptions

SerializationException SerializationException

The SerializationInfo object associated with the current OrderedDictionary collection is invalid.

Remarks

This method can be overridden.

OrderedDictionary OrderedDictionary

In this Article

Overloads

OrderedDictionary()	Initializes a new instance of the OrderedDictionary class.
OrderedDictionary(IEqualityComparer) OrderedDictionary(IEqualityComparer)	Initializes a new instance of the OrderedDictionary class using the specified comparer.
OrderedDictionary(Int32) OrderedDictionary(Int32)	Initializes a new instance of the OrderedDictionary class using the specified initial capacity.
OrderedDictionary(Int32, IEqualityComparer) Ordered Dictionary(Int32, IEqualityComparer)	Initializes a new instance of the OrderedDictionary class using the specified initial capacity and comparer.
OrderedDictionary(SerializationInfo, StreamingContext) OrderedDictionary(SerializationInfo, StreamingContext)	Initializes a new instance of the OrderedDictionary class that is serializable using the specified SerializationInfo and StreamingContext objects.

OrderedDictionary()

Initializes a new instance of the OrderedDictionary class.

```
public OrderedDictionary ();
```

Examples

The following code example demonstrates the creation and population of an OrderedDictionary collection. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Creates and initializes a OrderedDictionary.
OrderedDictionary myOrderedDictionary = new OrderedDictionary();
myOrderedDictionary.Add("testKey1", "testValue1");
myOrderedDictionary.Add("testKey2", "testValue2");
myOrderedDictionary.Add("keyToDelete", "valueToDelete");
myOrderedDictionary.Add("testKey3", "testValue3");

ICollection keyCollection = myOrderedDictionary.Keys;
ICollection valueCollection = myOrderedDictionary.Values;

// Display the contents using the key and value collections
DisplayContents(keyCollection, valueCollection, myOrderedDictionary.Count);
```

Remarks

The comparer determines whether two keys are equal. Every key in a OrderedDictionary collection must be unique. The default comparer is the key's implementation of Object.Equals.

OrderedDictionary(IEqualityComparer)

OrderedDictionary(IEqualityComparer)

Initializes a new instance of the OrderedDictionary class using the specified comparer.

```
public OrderedDictionary (System.Collections.IEqualityComparer comparer);
```

```
new System.Collections.Specialized.OrderedDictionary : System.Collections.IEqualityComparer ->
System.Collections.Specialized.OrderedDictionary
```

Parameters

comparer IEqualityComparer IEqualityComparer

The IComparer 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).

Remarks

The comparer determines whether two keys are equal. Every key in a OrderedDictionary collection must be unique. The default comparer is the key's implementation of Object.Equals.

The custom comparer enables such scenarios as doing lookups with case-insensitive strings.

OrderedDictionary(Int32) OrderedDictionary(Int32)

Initializes a new instance of the OrderedDictionary class using the specified initial capacity.

```
public OrderedDictionary (int capacity);
```

```
new System.Collections.Specialized.OrderedDictionary : int ->
System.Collections.Specialized.OrderedDictionary
```

Parameters

capacity Int32 Int32

The initial number of elements that the OrderedDictionary collection can contain.

Remarks

The comparer determines whether two keys are equal. Every key in a OrderedDictionary collection must be unique. The default comparer is the key's implementation of Object.Equals.

OrderedDictionary(Int32, IEqualityComparer) OrderedDictionary(Int32, IEqualityComparer)

Initializes a new instance of the OrderedDictionary class using the specified initial capacity and comparer.

```
public OrderedDictionary (int capacity, System.Collections.IEqualityComparer comparer);
```

```
new System.Collections.Specialized.OrderedDictionary : int * System.Collections.IEqualityComparer ->
System.Collections.Specialized.OrderedDictionary
```

Parameters

capacity Int32 Int32

The initial number of elements that the OrderedDictionary collection can contain.

comparer IEqualityComparer IEqualityComparer

The IComparer 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).

Remarks

The comparer determines whether two keys are equal. Every key in a OrderedDictionary collection must be unique. The default comparer is the key's implementation of Object.Equals.

The custom comparer enables such scenarios as doing lookups with case-insensitive strings.

OrderedDictionary(SerializationInfo, StreamingContext) OrderedDictionary(SerializationInfo, StreamingContext)

Initializes a new instance of the OrderedDictionary class that is serializable using the specified SerializationInfo and StreamingContext objects.

```
protected OrderedDictionary (System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context);
```

```
new System.Collections.Specialized.OrderedDictionary :
System.Runtime.Serialization.SerializationInfo * System.Runtime.Serialization.StreamingContext ->
System.Collections.Specialized.OrderedDictionary
```

Parameters

info SerializationInfo SerializationInfo

A SerializationInfo object containing the information required to serialize the OrderedDictionary collection.

context StreamingContext StreamingContext

A StreamingContext object containing the source and destination of the serialized stream associated with the OrderedDictionary.

Remarks

The comparer determines whether two keys are equal. Every key in a OrderedDictionary collection must be unique. The default comparer is the key's implementation of Object.Equals.

OrderedDictionary.Remove OrderedDictionary.Remove

In this Article

Removes the entry with the specified key from the OrderedDictionary collection.

```
public void Remove (object key);
```

```
abstract member Remove : obj -> unit
override this.Remove : obj -> unit
```

Parameters

key Object Object

The key of the entry to remove.

Exceptions

NotSupportedException NotSupportedException

The OrderedDictionary collection is read-only.

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the Remove method is used to remove the entry with the key `"keyToDelete"` from the OrderedDictionary. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Modifying the OrderedDictionary
if (!myOrderedDictionary.IsReadOnly)
{
    // Insert a new key to the beginning of the OrderedDictionary
    myOrderedDictionary.Insert(0, "insertedKey1", "insertedValue1");

    // Modify the value of the entry with the key "testKey2"
    myOrderedDictionary["testKey2"] = "modifiedValue";

    // Remove the last entry from the OrderedDictionary: "testKey3"
    myOrderedDictionary.RemoveAt(myOrderedDictionary.Count - 1);

    // Remove the "keyToDelete" entry, if it exists
    if (myOrderedDictionary.Contains("keyToDelete"))
    {
        myOrderedDictionary.Remove("keyToDelete");
    }
}
```

Remarks

The entries that follow the removed entry move up to occupy the vacated spot and the indexes of the entries that move are also updated.

If the OrderedDictionary collection does not contain an entry with the specified key, the OrderedDictionary remains unchanged and no exception is thrown.

OrderedDictionary.RemoveAt OrderedDictionary.RemoveAt

In this Article

Removes the entry at the specified index from the OrderedDictionary collection.

```
public void RemoveAt (int index);
```

```
abstract member RemoveAt : int -> unit
override this.RemoveAt : int -> unit
```

Parameters

index Int32 Int32

The zero-based index of the entry to remove.

Exceptions

NotSupportedException NotSupportedException

The OrderedDictionary collection is read-only.

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than zero.

- or -

`index` is equal to or greater than Count.

Examples

The following code example demonstrates the modification of an OrderedDictionary collection. In this example, the RemoveAt method is used with the Count property to remove the last entry from the OrderedDictionary. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Modifying the OrderedDictionary
if (!myOrderedDictionary.IsReadOnly)
{
    // Insert a new key to the beginning of the OrderedDictionary
    myOrderedDictionary.Insert(0, "insertedKey1", "insertedValue1");

    // Modify the value of the entry with the key "testKey2"
    myOrderedDictionary["testKey2"] = "modifiedValue";

    // Remove the last entry from the OrderedDictionary: "testKey3"
    myOrderedDictionary.RemoveAt(myOrderedDictionary.Count - 1);

    // Remove the "keyToDelete" entry, if it exists
    if (myOrderedDictionary.Contains("keyToDelete"))
    {
        myOrderedDictionary.Remove("keyToDelete");
    }
}
```

Remarks

The entries that follow the removed entry move up to occupy the vacated spot and the indexes of the entries that move are also updated.

OrderedDictionary.Values OrderedDictionary.Values

In this Article

Gets an ICollection object containing the values in the OrderedDictionary collection.

```
public System.Collections.ICollection Values { get; }
```

```
member this.Values : System.Collections.ICollection
```

Returns
ICollection ICollection

An ICollection object containing the values in the OrderedDictionary collection.

Examples

The following code example demonstrates the creation and population of an OrderedDictionary collection, and then prints the contents to the console. In this example, the Keys and Values properties are passed to a method that displays the contents. This code is part of a larger code example that can be viewed at OrderedDictionary.

```
// Creates and initializes a OrderedDictionary.
OrderedDictionary myOrderedDictionary = new OrderedDictionary();
myOrderedDictionary.Add("testKey1", "testValue1");
myOrderedDictionary.Add("testKey2", "testValue2");
myOrderedDictionary.Add("keyToDelete", "valueToDelete");
myOrderedDictionary.Add("testKey3", "testValue3");

ICollection keyCollection = myOrderedDictionary.Keys;
ICollection valueCollection = myOrderedDictionary.Values;

// Display the contents using the key and value collections
DisplayContents(keyCollection, valueCollection, myOrderedDictionary.Count);
```

```
// Displays the contents of the OrderedDictionary from its keys and values
public static void DisplayContents(
    ICollection keyCollection, ICollection valueCollection, int dictionarySize)
{
    String[] myKeys = new String[dictionarySize];
    String[] myValues = new String[dictionarySize];
    keyCollection.CopyTo(myKeys, 0);
    valueCollection.CopyTo(myValues, 0);

    // Displays the contents of the OrderedDictionary
    Console.WriteLine("   INDEX KEY                       VALUE");
    for (int i = 0; i < dictionarySize; i++)
    {
        Console.WriteLine("   {0,-5} {1,-25} {2}",
            i, myKeys[i], myValues[i]);
    }
    Console.WriteLine();
}
```

Remarks

The returned ICollection object is not a static copy; instead, the ICollection refers back to the values in the original OrderedDictionary collection. Therefore, changes to the OrderedDictionary continue to be reflected in the ICollection.

StringCollection StringCollection Class

Represents a collection of strings.

Declaration

```
[Serializable]
public class StringCollection : System.Collections.IList
```

```
type StringCollection = class
    interface IList
    interface ICollection
    interface IEnumerable
```

Inheritance Hierarchy

`Object` `Object`

Remarks

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

String comparisons are case-sensitive.

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

Constructors

`StringCollection()`

`StringCollection()`

Initializes a new instance of the StringCollection class.

Properties

`Count`

`Count`

Gets the number of strings contained in the StringCollection.

`IsReadOnly`

`IsReadOnly`

Gets a value indicating whether the StringCollection is read-only.

`IsSynchronized`

`IsSynchronized`

Gets a value indicating whether access to the StringCollection 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 StringCollection.

Methods

`Add(String)`

`Add(String)`

Adds a string to the end of the StringCollection.

`AddRange(String[])`

`AddRange(String[])`

Copies the elements of a string array to the end of the StringCollection.

`Clear()`

`Clear()`

Removes all the strings from the StringCollection.

`Contains(String)`

`Contains(String)`

Determines whether the specified string is in the StringCollection.

`CopyTo(String[], Int32)`

`CopyTo(String[], Int32)`

Copies the entire StringCollection values to a one-dimensional array of strings, starting at the specified index of the target array.

`GetEnumerator()`

`GetEnumerator()`

Returns a StringEnumerator that iterates through the StringCollection.

`IndexOf(String)`

`IndexOf(String)`

Searches for the specified string and returns the zero-based index of the first occurrence within the

StringCollection.

`Insert(Int32, String)`

`Insert(Int32, String)`

Inserts a string into the StringCollection at the specified index.

`Remove(String)`

`Remove(String)`

Removes the first occurrence of a specific string from the StringCollection.

`RemoveAt(Int32)`

`RemoveAt(Int32)`

Removes the string at the specified index of the StringCollection.

`ICollection.CopyTo(Array, Int32)`

`ICollection.CopyTo(Array, Int32)`

Copies the entire StringCollection to a compatible one-dimensional Array, starting at the specified index of the target array.

`IEnumerable.GetEnumerator()`

`IEnumerable.GetEnumerator()`

Returns a IEnumerator that iterates through the StringCollection.

`IList.Add(Object)`

`IList.Add(Object)`

Adds an object to the end of the StringCollection.

`IList.Contains(Object)`

`IList.Contains(Object)`

Determines whether an element is in the StringCollection.

`IList.IndexOf(Object)`

`IList.IndexOf(Object)`

Searches for the specified Object and returns the zero-based index of the first occurrence within the entire StringCollection.

`IList.Insert(Int32, Object)`

`IList.Insert(Int32, Object)`

Inserts an element into the StringCollection at the specified index.

`IList.IsFixedSize`

`IList.IsFixedSize`

Gets a value indicating whether the StringCollection object has a fixed size.

`IList.IsReadOnly`

`IList.IsReadOnly`

Gets a value indicating whether the StringCollection object 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 StringCollection.

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 StringCollection, but derived classes can create their own synchronized versions of the StringCollection 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

StringCollection.Add StringCollection.Add

In this Article

Adds a string to the end of the StringCollection.

```
public int Add (string value);
```

```
member this.Add : string -> int
```

Parameters

value String String

The string to add to the end of the StringCollection. The value can be `null`.

Returns

Int32 Int32

The zero-based index at which the new element is inserted.

Examples

The following code example adds new elements to the StringCollection.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Adds a range of elements from an array to the end of the StringCollection.
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "After adding a range of elements:" );
      PrintValues( myCol );

      // Adds one element to the end of the StringCollection and inserts another at index 3.
      myCol.Add( "* white" );
      myCol.Insert( 3, "* gray" );

      Console.WriteLine( "After adding \"* white\" to the end and inserting \"* gray\" at index 3:"
);
      PrintValues( myCol );

   }

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

```
}

/*
This code produces the following output.

Initial contents of the StringCollection:

After adding a range of elements:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After adding "* white" to the end and inserting "* gray" at index 3:
   RED
   orange
   yellow
   * gray
   RED
   green
   blue
   RED
   indigo
   violet
   RED
   * white

*/
```

Remarks

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

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 Also	AddRange(String[])AddRange(String[]) IsReadOnlyIsReadOnly

StringCollection.AddRange StringCollection.AddRange

In this Article

Copies the elements of a string array to the end of the StringCollection.

```
public void AddRange (string[] value);
```

```
member this.AddRange : string[] -> unit
```

Parameters

value String[]

An array of strings to add to the end of the StringCollection. The array itself can not be `null` but it can contain elements that are `null`.

Exceptions
ArgumentNullException ArgumentNullException

`value` is `null`.

Examples

The following code example adds new elements to the StringCollection.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Adds a range of elements from an array to the end of the StringCollection.
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "After adding a range of elements:" );
      PrintValues( myCol );

      // Adds one element to the end of the StringCollection and inserts another at index 3.
      myCol.Add( "* white" );
      myCol.Insert( 3, "* gray" );

      Console.WriteLine( "After adding \"* white\" to the end and inserting \"* gray\" at index 3:"
);
      PrintValues( myCol );

   }

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

```
    }
}

/*
This code produces the following output.

Initial contents of the StringCollection:

After adding a range of elements:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After adding "* white" to the end and inserting "* gray" at index 3:
   RED
   orange
   yellow
   * gray
   RED
   green
   blue
   RED
   indigo
   violet
   RED
   * white

*/
```

Remarks

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

If the StringCollection 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 Also	Add(String)Add(String) IsReadOnlyIsReadOnly

StringCollection.Clear StringCollection.Clear

In this Article

Removes all the strings from the StringCollection.

```
public void Clear ();
```

```
abstract member Clear : unit -> unit
override this.Clear : unit -> unit
```

Examples

The following code example removes elements from the StringCollection.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Removes one element from the StringCollection.
      myCol.Remove( "yellow" );

      Console.WriteLine( "After removing \"yellow\":" );
      PrintValues( myCol );

      // Removes all occurrences of a value from the StringCollection.
      int i = myCol.IndexOf( "RED" );
      while ( i > -1 )  {
         myCol.RemoveAt( i );
         i = myCol.IndexOf( "RED" );
      }

      Console.WriteLine( "After removing all occurrences of \"RED\":" );
      PrintValues( myCol );

      // Clears the entire collection.
      myCol.Clear();

      Console.WriteLine( "After clearing the collection:" );
      PrintValues( myCol );

   }

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

}
```

```
/*
This code produces the following output.

Initial contents of the StringCollection:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After removing "yellow":
   RED
   orange
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After removing all occurrences of "RED":
   orange
   green
   blue
   indigo
   violet

After clearing the collection:

*/
```

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.

See Also

CountCount

StringCollection.Contains StringCollection.Contains

In this Article

Determines whether the specified string is in the StringCollection.

```
public bool Contains (string value);
```

```
member this.Contains : string -> bool
```

Parameters

value String String

The string to locate in the StringCollection. The value can be `null`.

Returns

Boolean Boolean

`true` if `value` is found in the StringCollection; otherwise, `false`.

Examples

The following code example searches the StringCollection for an element.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Checks whether the collection contains "orange" and, if so, displays its index.
      if ( myCol.Contains( "orange" ) )
         Console.WriteLine( "The collection contains \"orange\" at index {0}.", myCol.IndexOf(
"orange" ) );
      else
         Console.WriteLine( "The collection does not contain \"orange\"." );

   }

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

}

/*
This code produces the following output.

Initial contents of the StringCollection:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

The collection contains "orange" at index 1.

*/
```

Remarks

The Contains method can confirm the existence of a string before performing further operations.

This method determines equality by calling Object.Equals. String comparisons are case-sensitive.

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 Also	IndexOf(String)IndexOf(String) Performing Culture-Insensitive String Operations

StringCollection.CopyTo StringCollection.CopyTo

In this Article

Copies the entire StringCollection values to a one-dimensional array of strings, starting at the specified index of the target array.

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

```
member this.CopyTo : string[] * int -> unit
```

Parameters

array String[]

The one-dimensional array of strings that is the destination of the elements copied from StringCollection. 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 StringCollection is greater than the available space from `index` to the end of the destination `array`.

InvalidCastException InvalidCastException

The type of the source StringCollection cannot be cast automatically to the type of the destination `array`.

Examples

The following code example copies a StringCollection to an array.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );
```

```
        Console.WriteLine( "Initial contents of the StringCollection:" );
        PrintValues( myCol );

        // Copies the collection to a new array starting at index 0.
        String[] myArr2 = new String[myCol.Count];
        myCol.CopyTo( myArr2, 0 );

        Console.WriteLine( "The new array contains:" );
        for ( int i = 0; i < myArr2.Length; i++ )  {
           Console.WriteLine( "   [{0}] {1}", i, myArr2[i] );
        }
        Console.WriteLine();

     }

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

}

/*
This code produces the following output.

Initial contents of the StringCollection:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

The new array contains:
   [0] RED
   [1] orange
   [2] yellow
   [3] RED
   [4] green
   [5] blue
   [6] RED
   [7] indigo
   [8] violet
   [9] RED

*/
```

Remarks

The specified array must be of a compatible type.

The elements are copied to the Array in the same order in which the enumerator of the StringCollection iterates through the StringCollection.

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

See ArrayArray

Also | GetEnumerator()GetEnumerator()

StringCollection.Count StringCollection.Count

In this Article

Gets the number of strings contained in the StringCollection.

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

```
member this.Count : int
```

Returns
Int32 Int32

The number of strings contained in the StringCollection.

Examples

The following code example copies a StringCollection to an array.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Copies the collection to a new array starting at index 0.
      String[] myArr2 = new String[myCol.Count];
      myCol.CopyTo( myArr2, 0 );

      Console.WriteLine( "The new array contains:" );
      for ( int i = 0; i < myArr2.Length; i++ )  {
         Console.WriteLine( "   [{0}] {1}", i, myArr2[i] );
      }
      Console.WriteLine();

   }

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

}

/*
This code produces the following output.

Initial contents of the StringCollection:
   RED
   orange
   yellow
```

```
   RED
   green
   blue
   RED
   indigo
   violet
   RED

The new array contains:
   [0] RED
   [1] orange
   [2] yellow
   [3] RED
   [4] green
   [5] blue
   [6] RED
   [7] indigo
   [8] violet
   [9] RED

*/
```

Remarks

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

StringCollection.GetEnumerator StringCollection.GetEnumerator

In this Article

Returns a StringEnumerator that iterates through the StringCollection.

```
public System.Collections.Specialized.StringEnumerator GetEnumerator ();
```

```
member this.GetEnumerator : unit -> System.Collections.Specialized.StringEnumerator
```

Returns
StringEnumerator StringEnumerator

A StringEnumerator for the StringCollection.

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 Also
StringEnumeratorStringEnumerator
IEnumeratorIEnumerator

StringCollection.ICollection.CopyTo

In this Article

Copies the entire StringCollection 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 StringCollection. 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 StringCollection is greater than the available space from `index` to the end of the destination `array`.

InvalidCastException

The type of the source StringCollection 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.

StringCollection.IEnumerable.GetEnumerator

In this Article

Returns a IEnumerator that iterates through the StringCollection.

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

Returns
IEnumerator

A IEnumerator for the StringCollection.

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	StringEnumerator IEnumerator

StringCollection.IList.Add

In this Article

Adds an object to the end of the StringCollection.

```
int IList.Add (object value);
```

Parameters

value Object

The Object to be added to the end of the StringCollection. The value can be `null`.

Returns
Int32

The StringCollection index at which the `value` has been added.

Exceptions
NotSupportedException

The StringCollection is read-only.

-or-

The StringCollection has a fixed size.

Remarks

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

If Count already equals the capacity, the capacity of the StringCollection is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.

If Count is less than the capacity, 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 Also Count

StringCollection.IList.Contains

In this Article

Determines whether an element is in the StringCollection.

```
bool IList.Contains (object value);
```

Parameters

value Object

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

Returns
Boolean

`true` if `value` is found in the StringCollection; 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 `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 Also Performing Culture-Insensitive String Operations

StringCollection.IList.IndexOf

In this Article

Searches for the specified Object and returns the zero-based index of the first occurrence within the entire StringCollection.

```
int IList.IndexOf (object value);
```

Parameters

value Object

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

Returns

Int32

The zero-based index of the first occurrence of `value` within the entire StringCollection, if found; otherwise, -1.

Remarks

The StringCollection is searched forward starting at the first element and ending at the last element.

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 `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 Also

Performing Culture-Insensitive String Operations

StringCollection.IList.Insert

In this Article

Inserts an element into the StringCollection 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. The value can be `null`.

Exceptions

ArgumentOutOfRangeException

`index` is less than zero.

-or-

`index` is greater than Count.

NotSupportedException

The StringCollection is read-only.

-or-

The StringCollection has a fixed size.

Remarks

If Count already equals the capacity, the capacity of the StringCollection 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 StringCollection.

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 Also Count

StringCollection.IList.IsFixedSize

In this Article

Gets a value indicating whether the StringCollection object has a fixed size.

```
bool System.Collections.IList.IsFixedSize { get; }
```

Returns

Boolean

`true` if the StringCollection 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.

StringCollection.IList.IsReadOnly

In this Article

Gets a value indicating whether the StringCollection object is read-only.

```
bool System.Collections.IList.IsReadOnly { get; }
```

Returns

Boolean

`true` if the StringCollection 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.

StringCollection.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.

Remarks

Thismethod provides the ability to access a specific element in the collection by using the following syntax: `myCollection[index]`.

The C# language uses the this keyword to define the indexers instead of implementing the IList.Item[Int32] property. Visual Basic implements IList.Item[Int32] as a default property, which provides the same indexing functionality.

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

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

See Also Count

StringCollection.IList.Remove

In this Article

Removes the first occurrence of a specific object from the StringCollection.

```
void IList.Remove (object value);
```

Parameters

value Object

The Object to remove from the StringCollection. The value can be `null`.

Exceptions
NotSupportedException

The StringCollection is read-only.

-or-

The StringCollection has a fixed size.

Remarks

If the StringCollection does not contain the specified object, the StringCollection 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 determines equality by calling Object.Equals.

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

See Also
Count
Performing Culture-Insensitive String Operations

StringCollection.IndexOf StringCollection.IndexOf

In this Article

Searches for the specified string and returns the zero-based index of the first occurrence within the StringCollection.

```
public int IndexOf (string value);
```

```
member this.IndexOf : string -> int
```

Parameters

value String String

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

Returns

Int32 Int32

The zero-based index of the first occurrence of `value` in the StringCollection, if found; otherwise, -1.

Examples

The following code example searches the StringCollection for an element.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Checks whether the collection contains "orange" and, if so, displays its index.
      if ( myCol.Contains( "orange" ) )
         Console.WriteLine( "The collection contains \"orange\" at index {0}.", myCol.IndexOf(
"orange" ) );
      else
         Console.WriteLine( "The collection does not contain \"orange\"." );

   }

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

}

/*
This code produces the following output.

Initial contents of the StringCollection:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

The collection contains "orange" at index 1.

*/
```

Remarks

This method determines equality by calling Object.Equals. String comparisons are case-sensitive.

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 Also	Contains(String)Contains(String) Performing Culture-Insensitive String Operations

StringCollection.Insert StringCollection.Insert

In this Article

Inserts a string into the StringCollection at the specified index.

```
public void Insert (int index, string value);
```

```
member this.Insert : int * string -> unit
```

Parameters

index Int32 Int32

The zero-based index at which `value` is inserted.

value String String

The string to insert. The value can be `null`.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than zero.

-or-

`index` greater than Count.

Examples

The following code example adds new elements to the StringCollection.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Adds a range of elements from an array to the end of the StringCollection.
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "After adding a range of elements:" );
      PrintValues( myCol );

      // Adds one element to the end of the StringCollection and inserts another at index 3.
      myCol.Add( "* white" );
      myCol.Insert( 3, "* gray" );

      Console.WriteLine( "After adding \"* white\" to the end and inserting \"* gray\" at index 3:"
);
      PrintValues( myCol );
```

```
    }

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

}

/*
This code produces the following output.

Initial contents of the StringCollection:

After adding a range of elements:
    RED
    orange
    yellow
    RED
    green
    blue
    RED
    indigo
    violet
    RED

After adding "* white" to the end and inserting "* gray" at index 3:
    RED
    orange
    yellow
    * gray
    RED
    green
    blue
    RED
    indigo
    violet
    RED
    * white

*/
```

Remarks

Duplicate strings are allowed in StringCollection.

If `index` is equal to Count, `value` is added to the end of StringCollection.

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 Also

CountCount
Add(String)Add(String)

StringCollection.IsReadOnly StringCollection.IsReadOnly

In this Article

Gets a value indicating whether the StringCollection is read-only.

```
public bool IsReadOnly { get; }
```

```
member this.IsReadOnly : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Remarks

StringCollection 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.

A StringCollection instance is always writable.

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

StringCollection.IsSynchronized StringCollection.Is Synchronized

In this Article

Gets a value indicating whether access to the StringCollection is synchronized (thread safe).

```
public bool IsSynchronized { get; }
```

```
member this.IsSynchronized : bool
```

Returns
Boolean Boolean

This property always returns `false`.

Remarks

StringCollection implements the IsSynchronized property because it is required by the 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.

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration:

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

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

See Also

SyncRootSyncRoot

StringCollection.Item[Int32] StringCollection.Item[Int32]

In this Article

Gets or sets the element at the specified index.

```
public string this[int index] { get; set; }
```

```
member this.Item(int) : string with get, set
```

Parameters

index Int32 Int32

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

Returns

String String

The element at the specified 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]`.

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

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 Also CountCount

StringCollection.Remove StringCollection.Remove

In this Article

Removes the first occurrence of a specific string from the StringCollection.

```
public void Remove (string value);
```

```
member this.Remove : string -> unit
```

Parameters

value

String String

The string to remove from the StringCollection. The value can be `null`.

Examples

The following code example removes elements from the StringCollection.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Removes one element from the StringCollection.
      myCol.Remove( "yellow" );

      Console.WriteLine( "After removing \"yellow\":" );
      PrintValues( myCol );

      // Removes all occurrences of a value from the StringCollection.
      int i = myCol.IndexOf( "RED" );
      while ( i > -1 )  {
         myCol.RemoveAt( i );
         i = myCol.IndexOf( "RED" );
      }

      Console.WriteLine( "After removing all occurrences of \"RED\":" );
      PrintValues( myCol );

      // Clears the entire collection.
      myCol.Clear();

      Console.WriteLine( "After clearing the collection:" );
      PrintValues( myCol );

   }

   public static void PrintValues( IEnumerable myCol )  {
      foreach ( Object obj in myCol )
```

```
         Console.WriteLine( "   {0}", obj );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the StringCollection:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After removing "yellow":
   RED
   orange
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After removing all occurrences of "RED":
   orange
   green
   blue
   indigo
   violet

After clearing the collection:

*/
```

Remarks

Duplicate strings are allowed in StringCollection. Only the first occurrence is removed. To remove all occurrences of the specified string, use `RemoveAt(IndexOf(value))` repeatedly while IndexOf does not return -1.

If the StringCollection does not contain the specified object, the StringCollection 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 determines equality by calling Object.Equals. String comparisons are case-sensitive.

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

See Also

Performing Culture-Insensitive String Operations

StringCollection.RemoveAt StringCollection.RemoveAt

In this Article

Removes the string at the specified index of the StringCollection.

```
public void RemoveAt (int index);
```

```
abstract member RemoveAt : int -> unit
override this.RemoveAt : int -> unit
```

Parameters

index Int32 Int32

The zero-based index of the string to remove.

Exceptions

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than zero.

-or-

`index` is equal to or greater than Count.

Examples

The following code example removes elements from the StringCollection.

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

public class SamplesStringCollection  {

   public static void Main()  {

      // Creates and initializes a new StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "RED", "orange", "yellow", "RED", "green", "blue", "RED",
"indigo", "violet", "RED" };
      myCol.AddRange( myArr );

      Console.WriteLine( "Initial contents of the StringCollection:" );
      PrintValues( myCol );

      // Removes one element from the StringCollection.
      myCol.Remove( "yellow" );

      Console.WriteLine( "After removing \"yellow\":" );
      PrintValues( myCol );

      // Removes all occurrences of a value from the StringCollection.
      int i = myCol.IndexOf( "RED" );
      while ( i > -1 )  {
         myCol.RemoveAt( i );
         i = myCol.IndexOf( "RED" );
      }

      Console.WriteLine( "After removing all occurrences of \"RED\":" );
      PrintValues( myCol );
```

```
      // Clears the entire collection.
      myCol.Clear();

      Console.WriteLine( "After clearing the collection:" );
      PrintValues( myCol );

   }

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

}

/*
This code produces the following output.

Initial contents of the StringCollection:
   RED
   orange
   yellow
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After removing "yellow":
   RED
   orange
   RED
   green
   blue
   RED
   indigo
   violet
   RED

After removing all occurrences of "RED":
   orange
   green
   blue
   indigo
   violet

After clearing the collection:

*/
```

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.

StringCollection

In this Article

Initializes a new instance of the StringCollection class.

```
public StringCollection ();
```

Remarks

This constructor is an O(1) operation.

StringCollection.SyncRoot StringCollection.SyncRoot

In this Article

Gets an object that can be used to synchronize access to the StringCollection.

```
public object SyncRoot { get; }
```

```
member this.SyncRoot : obj
```

Returns
Object Object

An object that can be used to synchronize access to the StringCollection.

Remarks

Derived classes can provide their own synchronized version of the StringCollection using the SyncRoot property. The synchronizing code must perform operations on the SyncRoot of the StringCollection, not directly on the StringCollection. 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 StringCollection 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:

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

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

See Also IsSynchronizedIsSynchronized

StringDictionary StringDictionary Class

Implements a hash table with the key and the value strongly typed to be strings rather than objects.

Declaration

```
[Serializable]
public class StringDictionary : System.Collections.IEnumerable
```

```
type StringDictionary = class
    interface IEnumerable
```

Inheritance Hierarchy

`Object` `Object`

Remarks

A key cannot be `null`, but a value can.

The key is handled in a case-insensitive manner; it is translated to lowercase before it is used with the string dictionary.

In .NET Framework version 1.0, this class uses culture-sensitive string comparisons. However, in .NET Framework version 1.1 and later, this class uses CultureInfo.InvariantCulture when comparing strings. For more information about how culture affects comparisons and sorting, see Performing Culture-Insensitive String Operations.

Constructors

`StringDictionary()`

`StringDictionary()`

Initializes a new instance of the StringDictionary class.

Properties

`Count`

`Count`

Gets the number of key/value pairs in the StringDictionary.

`IsSynchronized`

`IsSynchronized`

Gets a value indicating whether access to the StringDictionary is synchronized (thread safe).

`Item[String]`

`Item[String]`

Gets or sets the value associated with the specified key.

`Keys`

`Keys`

Gets a collection of keys in the StringDictionary.

`SyncRoot`

`SyncRoot`

Gets an object that can be used to synchronize access to the StringDictionary.

`Values`

`Values`

Gets a collection of values in the StringDictionary.

Methods

`Add(String, String)`

`Add(String, String)`

Adds an entry with the specified key and value into the StringDictionary.

`Clear()`

`Clear()`

Removes all entries from the StringDictionary.

`ContainsKey(String)`

`ContainsKey(String)`

Determines if the StringDictionary contains a specific key.

`ContainsValue(String)`

`ContainsValue(String)`

Determines if the StringDictionary contains a specific value.

`CopyTo(Array, Int32)`

`CopyTo(Array, Int32)`

Copies the string dictionary values to a one-dimensional Array instance at the specified index.

`GetEnumerator()`

`GetEnumerator()`

Returns an enumerator that iterates through the string dictionary.

`Remove(String)`

`Remove(String)`

Removes the entry with the specified key from the string dictionary.

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 StringDictionary, but derived classes can create their own synchronized versions of the StringDictionary 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

StringDictionary.Add StringDictionary.Add

In this Article

Adds an entry with the specified key and value into the StringDictionary.

```
public virtual void Add (string key, string value);
```

```
abstract member Add : string * string -> unit
override this.Add : string * string -> unit
```

Parameters

key String String

The key of the entry to add.

value String String

The value of the entry to add. The value can be `null`.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

ArgumentException ArgumentException

An entry with the same key already exists in the StringDictionary.

NotSupportedException NotSupportedException

The StringDictionary is read-only.

Examples

The following code example demonstrates how to add and remove elements from a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Displays the values in the StringDictionary.
      Console.WriteLine( "Initial contents of the StringDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes an element.
      myCol.Remove( "green" );
      Console.WriteLine( "The collection contains the following elements after removing \"green\":"
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( StringDictionary myCol )  {
      Console.WriteLine( "   KEY        VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-10} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the StringDictionary:
   KEY        VALUE
   green      verde
   red        rojo
   blue       azul

The collection contains the following elements after removing "green":
   KEY        VALUE
   red        rojo
   blue       azul

The collection contains the following elements after it is cleared:
   KEY        VALUE

*/
```

Remarks

The key is handled in a case-insensitive manner; it is translated to lowercase before it is added to the string dictionary.

This method is an O(1) operation.

See Also	Remove(String)Remove(String)

StringDictionary.Clear StringDictionary.Clear

In this Article

Removes all entries from the StringDictionary.

```
public virtual void Clear ();
```

```
abstract member Clear : unit -> unit
override this.Clear : unit -> unit
```

Exceptions

NotSupportedException NotSupportedException

The StringDictionary is read-only.

Examples

The following code example demonstrates how to add and remove elements from a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Displays the values in the StringDictionary.
      Console.WriteLine( "Initial contents of the StringDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes an element.
      myCol.Remove( "green" );
      Console.WriteLine( "The collection contains the following elements after removing \"green\":"
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( StringDictionary myCol )  {
      Console.WriteLine( "   KEY        VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-10} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the StringDictionary:
   KEY        VALUE
   green      verde
   red        rojo
   blue       azul

The collection contains the following elements after removing "green":
   KEY        VALUE
   red        rojo
   blue       azul

The collection contains the following elements after it is cleared:
   KEY        VALUE

*/
```

Remarks

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

StringDictionary.ContainsKey StringDictionary.ContainsKey

In this Article

Determines if the StringDictionary contains a specific key.

```
public virtual bool ContainsKey (string key);
```

```
abstract member ContainsKey : string -> bool
override this.ContainsKey : string -> bool
```

Parameters

key String String

The key to locate in the StringDictionary.

Returns

Boolean Boolean

`true` if the StringDictionary contains an entry with the specified key; otherwise, `false`.

Exceptions

ArgumentNullException ArgumentNullException

The key is `null`.

Examples

The following code example searches for an element in a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Displays the values in the StringDictionary.
      Console.WriteLine( "Initial contents of the StringDictionary:" );
      PrintKeysAndValues( myCol );

      // Searches for a key.
      if ( myCol.ContainsKey( "red" ) )
         Console.WriteLine( "The collection contains the key \"red\"." );
      else
         Console.WriteLine( "The collection does not contain the key \"red\"." );
      Console.WriteLine();

      // Searches for a value.
      if ( myCol.ContainsValue( "amarillo" ) )
         Console.WriteLine( "The collection contains the value \"amarillo\"." );
      else
         Console.WriteLine( "The collection does not contain the value \"amarillo\"." );
      Console.WriteLine();

   }

   public static void PrintKeysAndValues( StringDictionary myCol )  {
      Console.WriteLine( "   KEY        VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-10} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the StringDictionary:
   KEY        VALUE
   green      verde
   red        rojo
   blue       azul

The collection contains the key "red".

The collection does not contain the value "amarillo".

*/
```

Remarks

The key is handled in a case-insensitive manner; it is translated to lowercase before it is used.

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 Also	Performing Culture-Insensitive String Operations

StringDictionary.ContainsValue StringDictionary.ContainsValue

In this Article

Determines if the StringDictionary contains a specific value.

```
public virtual bool ContainsValue (string value);
```

```
abstract member ContainsValue : string -> bool
override this.ContainsValue : string -> bool
```

Parameters

value String String

The value to locate in the StringDictionary. The value can be `null`.

Returns

Boolean Boolean

`true` if the StringDictionary contains an element with the specified value; otherwise, `false`.

Examples

The following code example searches for an element in a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Displays the values in the StringDictionary.
      Console.WriteLine( "Initial contents of the StringDictionary:" );
      PrintKeysAndValues( myCol );

      // Searches for a key.
      if ( myCol.ContainsKey( "red" ) )
         Console.WriteLine( "The collection contains the key \"red\"." );
      else
         Console.WriteLine( "The collection does not contain the key \"red\"." );
      Console.WriteLine();

      // Searches for a value.
      if ( myCol.ContainsValue( "amarillo" ) )
         Console.WriteLine( "The collection contains the value \"amarillo\"." );
      else
         Console.WriteLine( "The collection does not contain the value \"amarillo\"." );
      Console.WriteLine();

   }

   public static void PrintKeysAndValues( StringDictionary myCol )  {
      Console.WriteLine( "   KEY        VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-10} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the StringDictionary:
   KEY        VALUE
   green      verde
   red        rojo
   blue       azul

The collection contains the key "red".

The collection does not contain the value "amarillo".

*/
```

Remarks

The values of the elements of the StringDictionary are compared to the specified value using the Object.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 Also	Performing Culture-Insensitive String Operations

StringDictionary.CopyTo StringDictionary.CopyTo

In this Article

Copies the string dictionary values to a one-dimensional Array instance at the specified 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 values copied from the StringDictionary.

index Int32 Int32

The index in the array where copying begins.

Exceptions

ArgumentException ArgumentException

`array` is multidimensional.

-or-

The number of elements in the StringDictionary is greater than the available space from `index` to the end of `array`.

ArgumentNullException ArgumentNullException

`array` is `null`.

ArgumentOutOfRangeException ArgumentOutOfRangeException

`index` is less than the lower bound of `array`.

Examples

The following code example shows how a StringDictionary can be copied to an array.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Displays the values in the StringDictionary.
      Console.WriteLine( "KEYS    VALUES in the StringDictionary" );
      foreach ( DictionaryEntry myDE in myCol )
         Console.WriteLine( "{0}    {1}", myDE.Key, myDE.Value );
      Console.WriteLine();

      // Creates an array with DictionaryEntry elements.
      DictionaryEntry[] myArr = { new DictionaryEntry(), new DictionaryEntry(), new
DictionaryEntry() };

      // Copies the StringDictionary to the array.
      myCol.CopyTo( myArr, 0 );

      // Displays the values in the array.
      Console.WriteLine( "KEYS    VALUES in the array" );
      for ( int i = 0; i < myArr.Length; i++ )
         Console.WriteLine( "{0}    {1}", myArr[i].Key, myArr[i].Value );
      Console.WriteLine();

   }

}

/*
This code produces the following output.

KEYS    VALUES in the StringDictionary
green   verde
red     rojo
blue    azul

KEYS    VALUES in the array
green   verde
red     rojo
blue    azul

*/
```

Remarks

CopyTo copies objects that can be typecast to System.Collections.DictionaryEntry. DictionaryEntry contains both the key and the value.

The elements copied to the Array are sorted in the same order that the enumerator iterates through the StringDictionary.

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

StringDictionary.Count StringDictionary.Count

In this Article

Gets the number of key/value pairs in the StringDictionary.

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

```
member this.Count : int
```

Returns
Int32 Int32

The number of key/value pairs in the StringDictionary.

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

Examples

The following code example enumerates the elements of a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( StringDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( StringDictionary myCol )  {
      IEnumerator myEnumerator = myCol.GetEnumerator();
```

```
      DictionaryEntry de;
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )  {
         de = (DictionaryEntry) myEnumerator.Current;
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      }
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( StringDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the IEnumerator:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     red                       rojo
   1     blue                      azul
   2     green                     verde

*/
```

StringDictionary.GetEnumerator StringDictionary.GetEnumerator

In this Article

Returns an enumerator that iterates through the string dictionary.

```
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 that iterates through the string dictionary.

Examples

The following code example enumerates the elements of a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( StringDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( StringDictionary myCol )  {
```

```
      IEnumerator myEnumerator = myCol.GetEnumerator();
      DictionaryEntry de;
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )  {
         de = (DictionaryEntry) myEnumerator.Current;
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      }
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( StringDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the IEnumerator:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     red                       rojo
   1     blue                      azul
   2     green                     verde

*/
```

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.

StringDictionary.IsSynchronized StringDictionary.Is Synchronized

In this Article

Gets a value indicating whether access to the StringDictionary is synchronized (thread safe).

```
public virtual bool IsSynchronized { get; }
```

```
member this.IsSynchronized : bool
```

Returns

Boolean Boolean

`true` if access to the StringDictionary is synchronized (thread safe); otherwise, `false`.

Examples

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration.

```
StringDictionary myCollection = new StringDictionary();
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 StringDictionary instance is not synchronized. Derived classes can provide a synchronized version of the StringDictionary 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.

StringDictionary.Item[String] StringDictionary.Item[String]

In this Article

Gets or sets the value associated with the specified key.

```
public virtual string this[string key] { get; set; }
```

```
member this.Item(string) : string with get, set
```

Parameters

key String String

The key whose value to get or set.

Returns

String String

The value associated with the specified key. If the specified key is not found, Get returns `null`, and Set creates a new entry with the specified key.

Exceptions

ArgumentNullException ArgumentNullException

`key` is `null`.

Examples

The following code example enumerates the elements of a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }
```

```
   // 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( StringDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( StringDictionary myCol )  {
      IEnumerator myEnumerator = myCol.GetEnumerator();
      DictionaryEntry de;
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )  {
         de = (DictionaryEntry) myEnumerator.Current;
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      }
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( StringDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the IEnumerator:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     red                       rojo
   1     blue                      azul
   2     green                     verde

*/
```

Remarks

The key is handled in a case-insensitive manner; it is translated to lowercase before it is used.

A key cannot be `null`, but a value can. 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 ContainsKey method to determine if the key exists in the list.

The C# language uses the keyword to define the indexers instead of implementing the Item[String] property. Visual Basic implements Item[String] 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 Also	Performing Culture-Insensitive String Operations

StringDictionary.Keys StringDictionary.Keys

In this Article

Gets a collection of keys in the StringDictionary.

```
public virtual System.Collections.ICollection Keys { get; }
```

```
member this.Keys : System.Collections.ICollection
```

Returns
ICollection ICollection

An ICollection that provides the keys in the StringDictionary.

Examples

The following code example enumerates the elements of a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

   }

   // 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( StringDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( StringDictionary myCol )  {
      IEnumerator myEnumerator = myCol.GetEnumerator();
      DictionaryEntry de;
      Console.WriteLine( "   KEY                       VALUE" );
```

```
      while ( myEnumerator.MoveNext() )  {
         de = (DictionaryEntry) myEnumerator.Current;
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      }
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( StringDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the IEnumerator:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     red                       rojo
   1     blue                      azul
   2     green                     verde

*/
```

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 StringDictionary. Therefore, changes to the StringDictionary continue to be reflected in the ICollection.

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

StringDictionary.Remove StringDictionary.Remove

In this Article

Removes the entry with the specified key from the string dictionary.

```
public virtual void Remove (string key);
```

```
abstract member Remove : string -> unit
override this.Remove : string -> unit
```

Parameters

key String String

The key of the entry to remove.

Exceptions

ArgumentNullException ArgumentNullException

The key is `null`.

NotSupportedException NotSupportedException

The StringDictionary is read-only.

Examples

The following code example demonstrates how to add and remove elements from a StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Displays the values in the StringDictionary.
      Console.WriteLine( "Initial contents of the StringDictionary:" );
      PrintKeysAndValues( myCol );

      // Deletes an element.
      myCol.Remove( "green" );
      Console.WriteLine( "The collection contains the following elements after removing \"green\":"
);
      PrintKeysAndValues( myCol );

      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues( myCol );

   }

   public static void PrintKeysAndValues( StringDictionary myCol )  {
      Console.WriteLine( "   KEY        VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-10} {1}", de.Key, de.Value );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Initial contents of the StringDictionary:
   KEY        VALUE
   green      verde
   red        rojo
   blue       azul

The collection contains the following elements after removing "green":
   KEY        VALUE
   red        rojo
   blue       azul

The collection contains the following elements after it is cleared:
   KEY        VALUE

*/
```

Remarks

If the StringDictionary does not contain an element with the specified key, the StringDictionary remains unchanged. No exception is thrown.

The key is handled in a case-insensitive manner; it is translated to lowercase before it is used to find the entry to remove from the string dictionary.

This method is an O(1) operation.

See Also	Add(String, String)Add(String, String) Performing Culture-Insensitive String Operations

StringDictionary

In this Article

Initializes a new instance of the StringDictionary class.

```
public StringDictionary ();
```

Examples

The following code example demonstrates several of the properties and methods of StringDictionary.

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

public class SamplesStringDictionary  {

   public static void Main()  {

      // Creates and initializes a new StringDictionary.
      StringDictionary myCol = new StringDictionary();
      myCol.Add( "red", "rojo" );
      myCol.Add( "green", "verde" );
      myCol.Add( "blue", "azul" );

      // Display the contents of the collection using foreach. This is the preferred method.
      Console.WriteLine( "Displays the elements using foreach:" );
      PrintKeysAndValues1( myCol );

      // Display the contents of the collection using the enumerator.
      Console.WriteLine( "Displays the elements using the IEnumerator:" );
      PrintKeysAndValues2( myCol );

      // Display the contents of the collection using the Keys, Values, Count, and Item properties.
      Console.WriteLine( "Displays the elements using the Keys, Values, Count, and Item properties:"
);
      PrintKeysAndValues3( myCol );

      // Copies the StringDictionary to an array with DictionaryEntry elements.
      DictionaryEntry[] myArr = new DictionaryEntry[myCol.Count];
      myCol.CopyTo( myArr, 0 );

      // Displays the values in the array.
      Console.WriteLine( "Displays the elements in the array:" );
      Console.WriteLine( "   KEY        VALUE" );
      for ( int i = 0; i < myArr.Length; i++ )
         Console.WriteLine( "   {0,-10} {1}", myArr[i].Key, myArr[i].Value );
      Console.WriteLine();

      // Searches for a value.
      if ( myCol.ContainsValue( "amarillo" ) )
         Console.WriteLine( "The collection contains the value \"amarillo\"." );
      else
         Console.WriteLine( "The collection does not contain the value \"amarillo\"." );
      Console.WriteLine();

      // Searches for a key and deletes it.
      if ( myCol.ContainsKey( "green" ) )
         myCol.Remove( "green" );
      Console.WriteLine( "The collection contains the following elements after removing \"green\":"
);
      PrintKeysAndValues1( myCol );
```

```
      // Clears the entire collection.
      myCol.Clear();
      Console.WriteLine( "The collection contains the following elements after it is cleared:" );
      PrintKeysAndValues1( myCol );

   }

   // 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( StringDictionary myCol )  {
      Console.WriteLine( "   KEY                       VALUE" );
      foreach ( DictionaryEntry de in myCol )
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.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( StringDictionary myCol )  {
      IEnumerator myEnumerator = myCol.GetEnumerator();
      DictionaryEntry de;
      Console.WriteLine( "   KEY                       VALUE" );
      while ( myEnumerator.MoveNext() )  {
         de = (DictionaryEntry) myEnumerator.Current;
         Console.WriteLine( "   {0,-25} {1}", de.Key, de.Value );
      }
      Console.WriteLine();
   }

   // Uses the Keys, Values, Count, and Item properties.
   public static void PrintKeysAndValues3( StringDictionary myCol )  {
      String[] myKeys = new String[myCol.Count];
      myCol.Keys.CopyTo( myKeys, 0 );

      Console.WriteLine( "   INDEX KEY                       VALUE" );
      for ( int i = 0; i < myCol.Count; i++ )
         Console.WriteLine( "   {0,-5} {1,-25} {2}", i, myKeys[i], myCol[myKeys[i]] );
      Console.WriteLine();
   }

}

/*
This code produces the following output.

Displays the elements using foreach:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the IEnumerator:
   KEY                       VALUE
   red                       rojo
   blue                      azul
   green                     verde

Displays the elements using the Keys, Values, Count, and Item properties:
   INDEX KEY                       VALUE
   0     red                       rojo
   1     blue                      azul
   2     green                     verde

Displays the elements in the array:
   KEY        VALUE
```

```
   KEY        VALUE
   red        rojo
   blue       azul
   green      verde

The collection does not contain the value "amarillo".

The collection contains the following elements after removing "green":
   KEY                       VALUE
   red                       rojo
   blue                      azul

The collection contains the following elements after it is cleared:
   KEY                       VALUE

*/
```

Remarks

This constructor is an O(1) operation.

StringDictionary.SyncRoot StringDictionary.SyncRoot

In this Article

Gets an object that can be used to synchronize access to the StringDictionary.

```
public virtual object SyncRoot { get; }
```

```
member this.SyncRoot : obj
```

Returns

Object Object

An Object that can be used to synchronize access to the StringDictionary.

Examples

The following code example shows how to lock the collection using the SyncRoot during the entire enumeration.

```
StringDictionary myCollection = new StringDictionary();
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 StringDictionary using the SyncRoot property. The synchronizing code must perform operations on the SyncRoot of the StringDictionary, not directly on the StringDictionary. 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 StringDictionary 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.

StringDictionary.Values StringDictionary.Values

In this Article

Gets a collection of values in the StringDictionary.

```
public virtual System.Collections.ICollection Values { get; }
```

```
member this.Values : System.Collections.ICollection
```

Returns
ICollection ICollection

An ICollection that provides the values in the StringDictionary.

Examples

The following code example enumerates the elements of a StringDictionary.

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

public class SamplesStringDictionary
{
    public static void Main()
    {
        // Creates and initializes a new StringDictionary.
        StringDictionary myCol = new StringDictionary();
        myCol.Add( "red", "rojo" );
        myCol.Add( "green", "verde" );
        myCol.Add( "blue", "azul" );

        Console.WriteLine("VALUES");
        foreach (string val in myCol.Values)
        {
            Console.WriteLine(val);
        }
    }
}
// This code produces the following output.
// VALUE
// verde
// rojo
// azul
```

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 StringDictionary. Therefore, changes to the StringDictionary continue to be reflected in the ICollection.

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

StringEnumerator StringEnumerator Class

Supports a simple iteration over a StringCollection.

Declaration

```
public class StringEnumerator
```

```
type StringEnumerator = class
```

Inheritance Hierarchy

`Object` `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, 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.

Properties

`Current`

`Current`

Gets the current element in the collection.

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.

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.

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.

StringEnumerator.Current StringEnumerator.Current

In this Article

Gets the current element in the collection.

```
public string Current { get; }
```

```
member this.Current : string
```

Returns

String String

The current element in the collection.

Exceptions

InvalidOperationException InvalidOperationException

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

Examples

The following code example demonstrates several of the properties and methods of StringEnumerator.

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

public class SamplesStringEnumerator  {

   public static void Main()  {

      // Creates and initializes a StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "red", "orange", "yellow", "green", "blue", "indigo", "violet"
};
      myCol.AddRange( myArr );

      // Enumerates the elements in the StringCollection.
      StringEnumerator myEnumerator = myCol.GetEnumerator();
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "{0}", myEnumerator.Current );
      Console.WriteLine();

      // Resets the enumerator and displays the first element again.
      myEnumerator.Reset();
      if ( myEnumerator.MoveNext() )
         Console.WriteLine( "The first element is {0}.", myEnumerator.Current );

   }

}

/*
This code produces the following output.

red
orange
yellow
green
blue
indigo
violet

The first element is red.

*/
```

Remarks

After an enumerator is created or after a Reset is called, MoveNext must be called to advance the enumerator to the first element of the collection before reading the value of Current; otherwise, Current is undefined.

Current also throws an exception if the last call to MoveNext returned `false`, which indicates the end of the collection.

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

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.

See Also	MoveNext()MoveNext() Reset()Reset()

StringEnumerator.MoveNext StringEnumerator.MoveNext

In this Article

Advances the enumerator to the next element of the collection.

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

```
member this.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.

Exceptions

InvalidOperationException InvalidOperationException

The collection was modified after the enumerator was created.

Examples

The following code example demonstrates several of the properties and methods of StringEnumerator.

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

public class SamplesStringEnumerator  {

   public static void Main()  {

      // Creates and initializes a StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "red", "orange", "yellow", "green", "blue", "indigo", "violet"
};
      myCol.AddRange( myArr );

      // Enumerates the elements in the StringCollection.
      StringEnumerator myEnumerator = myCol.GetEnumerator();
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "{0}", myEnumerator.Current );
      Console.WriteLine();

      // Resets the enumerator and displays the first element again.
      myEnumerator.Reset();
      if ( myEnumerator.MoveNext() )
         Console.WriteLine( "The first element is {0}.", myEnumerator.Current );

   }

}

/*
This code produces the following output.

red
orange
yellow
green
blue
indigo
violet

The first element is red.

*/
```

Remarks

After an enumerator is created or after a Reset is called, an enumerator is positioned before the first element of the collection, and the first call to MoveNext 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.

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.

See Also	CurrentCurrent Reset()Reset()

StringEnumerator.Reset StringEnumerator.Reset

In this Article

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

```
public void Reset ();
```

```
member this.Reset : unit -> unit
```

Exceptions

InvalidOperationException InvalidOperationException

The collection was modified after the enumerator was created.

Examples

The following code example demonstrates several of the properties and methods of StringEnumerator.

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

public class SamplesStringEnumerator  {

   public static void Main()  {

      // Creates and initializes a StringCollection.
      StringCollection myCol = new StringCollection();
      String[] myArr = new String[] { "red", "orange", "yellow", "green", "blue", "indigo", "violet"
};
      myCol.AddRange( myArr );

      // Enumerates the elements in the StringCollection.
      StringEnumerator myEnumerator = myCol.GetEnumerator();
      while ( myEnumerator.MoveNext() )
         Console.WriteLine( "{0}", myEnumerator.Current );
      Console.WriteLine();

      // Resets the enumerator and displays the first element again.
      myEnumerator.Reset();
      if ( myEnumerator.MoveNext() )
         Console.WriteLine( "The first element is {0}.", myEnumerator.Current );

   }

}

/*
This code produces the following output.

red
orange
yellow
green
blue
indigo
violet

The first element is red.

*/
```

Remarks

Reset moves the enumerator to the beginning of the collection, before the first element. After Reset, MoveNext must be called to advance the enumerator to the first element of the collection before reading the value of Current.

See Also	MoveNext()MoveNext() CurrentCurrent