Microsoft

CLR via C#

Fourth Edition

Developer Reference

Jeffrey Richter

Wintellect®
Know how.

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2012951989
ISBN: 978-0-7356-6745-7

Printed and bound in the United States of America.

Third Printing

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Carol Dillingham
Editorial Production: Online Training Solutions, Inc.
Technical Reviewer: Christophe Nasarre; Technical Review services provided by Content Master,
a member of CM Group, Ltd.
Copyeditor: Candace Sinclair
Indexer: Jan Bednarczuk
Cover: Twist Creative • Seattle

[LSI]
[2013-05-17]

Note In addition to anonymous types and the Tuple types, you might want to take a look at the System.Dynamic.ExpandoObject class (defined in the System.Core.dll assembly). When you use this class with C#'s dynamic type (discussed in Chapter 5, "Primitive, Reference, and Value Types"), you have another way of grouping a set of properties (key/value pairs) together. The result is not compile-time type-safe, but the syntax looks nice (although you get no IntelliSense support), and you can pass ExpandoObject objects between C# and dynamic languages like Python. Here's some sample code that uses an ExpandoObject.

```
dynamic e = new System.Dynamic.ExpandoObject();
e.x = 6; // Add an Int32 'x' property whose value is 6
e.y = "Jeff";    // Add a String 'y' property whose value is "Jeff"
e.z = null;      // Add an Object 'z' property whose value is null

// See all the properties and their values:
foreach (var v in (IDictionary<String, Object>)e)
   Console.WriteLine("Key={0}, V={1}", v.Key, v.Value);


// Remove the 'x' property and its value
var d = (IDictionary<String, Object>)e;
d.Remove("x");
```

Parameterful Properties

In the previous section, the get accessor methods for the properties accepted no parameters. For this reason, I called these properties *parameterless properties*. These properties are easy to understand because they have the feel of accessing a field. In addition to these field-like properties, programming languages also support what I call *parameterful properties*, whose get accessor methods accept one or more parameters and whose set accessor methods accept two or more parameters. Different programming languages expose parameterful properties in different ways. Also, languages use different terms to refer to parameterful properties: C# calls them indexers and Visual Basic calls them *default properties*. In this section, I'll focus on how C# exposes its indexers by using parameterful properties.

In C#, parameterful properties (indexers) are exposed using an array-like syntax. In other words, you can think of an indexer as a way for the C# developer to overload the [] operator. Here's an example of a BitArray class that allows array-like syntax to index into the set of bits maintained by an instance of the class.

```
using System;

public sealed class BitArray {
   // Private array of bytes that hold the bits
   private Byte[] m_byteArray;
   private Int32  m_numBits;

   // Constructor that allocates the byte array and sets all bits to 0
   public BitArray(Int32 numBits) {
      // Validate arguments first.
      if (numBits <= 0)
         throw new ArgumentOutOfRangeException("numBits must be > 0");

      // Save the number of bits.
      m_numBits = numBits;

      // Allocate the bytes for the bit array.
      m_byteArray = new Byte[(numBits + 7) / 8];
   }

   // This is the indexer (parameterful property).
   public Boolean this[Int32 bitPos] {

      // This is the indexer's get accessor method.
      get {
         // Validate arguments first
         if ((bitPos < 0) || (bitPos >= m_numBits))
            throw new ArgumentOutOfRangeException("bitPos");

         // Return the state of the indexed bit.
         return (m_byteArray[bitPos / 8] & (1 << (bitPos % 8))) != 0;
      }

      // This is the indexer's set accessor method.
      set {
         if ((bitPos < 0) || (bitPos >= m_numBits))
            throw new ArgumentOutOfRangeException("bitPos", bitPos.ToString());
         if (value) {
            // Turn the indexed bit on.
            m_byteArray[bitPos / 8] = (Byte)
               (m_byteArray[bitPos / 8] | (1 << (bitPos % 8)));
         } else {
            // Turn the indexed bit off.
            m_byteArray[bitPos / 8] = (Byte)
               (m_byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
         }
      }
   }
}
```

Using the BitArray class's indexer is incredibly simple.

```
// Allocate a BitArray that can hold 14 bits.
BitArray ba = new BitArray(14);

// Turn all the even-numbered bits on by calling the set accessor.
for (Int32 x = 0; x < 14; x++) {
   ba[x] = (x % 2 == 0);
}

// Show the state of all the bits by calling the get accessor.
for (Int32 x = 0; x < 14; x++) {
   Console.WriteLine("Bit " + x + " is " + (ba[x] ? "On" : "Off"));
}
```

In the BitArray example, the indexer takes one Int32 parameter, bitPos. All indexers must have at least one parameter, but they can have more. These parameters (as well as the return type) can be of any data type (except void). An example of an indexer that has more than one parameter can be found in the System.Drawing.Imaging.ColorMatrix class, which ships in the System.Drawing.dll assembly.

It's quite common to create an indexer to look up values in an associative array. In fact, the System.Collections.Generic.Dictionary type offers an indexer that takes a key and returns the value associated with the key. Unlike parameterless properties, a type can offer multiple, overloaded indexers as long as their signatures differ.

Like a parameterless property's set accessor method, an indexer's set accessor method also contains a hidden parameter, called value in C#. This parameter indicates the new value desired for the "indexed element."

The CLR doesn't differentiate parameterless properties and parameterful properties; to the CLR, each is simply a pair of methods and a piece of metadata defined within a type. As mentioned earlier, different programming languages require different syntax to create and use parameterful properties. The fact that C# requires this[...] as the syntax for expressing an indexer was purely a choice made by the C# team. What this choice means is that C# allows indexers to be defined only on instances of objects. C# doesn't offer syntax allowing a developer to define a static indexer property, although the CLR does support static parameterful properties.

Because the CLR treats parameterful properties just as it does parameterless properties, the compiler will emit either two or three of the following items into the resulting managed assembly:

- A method representing the parameterful property's get accessor method. This is emitted only if you define a get accessor method for the property.
- A method representing the parameterful property's set accessor method. This is emitted only if you define a set accessor method for the property.
- A property definition in the managed assembly's metadata, which is always emitted. There's no special parameterful property metadata definition table because, to the CLR, parameterful properties are just properties.

For the BitArray class shown earlier, the compiler compiles the indexer as though the original source code were written as follows.

```
public sealed class BitArray {

   // This is the indexer's get accessor method.
   public Boolean get_Item(Int32 bitPos) { /* ... */ }

   // This is the indexer's set accessor method.
   public void    set_Item(Int32 bitPos, Boolean value)  { /* ... */ }
}
```

The compiler automatically generates names for these methods by prepending get_ and set_ to the *indexer name*. Because the C# syntax for an indexer doesn't allow the developer to specify an *indexer name*, the C# compiler team had to choose a default name to use for the accessor methods; they chose Item. Therefore, the method names emitted by the compiler are get_Item and set_Item.

When examining the .NET Framework Reference documentation, you can tell if a type offers an indexer by looking for a property named Item. For example, the System.Collections.Generic.List type offers a public instance property named Item; this property is List's indexer.

When you program in C#, you never see the name of Item, so you don't normally care that the compiler has chosen this name for you. However, if you're designing an indexer for a type that code written in other programming languages will be accessing, you might want to change the default name, Item, given to your indexer's get and set accessor methods. C# allows you to rename these methods by applying the System.Runtime.CompilerServices.IndexerNameAttribute custom attribute to the indexer. The following code demonstrates how to do this.

```
using System;
using System.Runtime.CompilerServices;

public sealed class BitArray {

   [IndexerName("Bit")]
   public Boolean this[Int32 bitPos] {
      // At least one accessor method is defined here
   }
}
```

Now the compiler will emit methods called get_Bit and set_Bit instead of get_Item and set_Item. When compiling, the C# compiler sees the IndexerName attribute, and this tells the compiler how to name the methods and the property metadata; the attribute itself is not emitted into the assembly's metadata.[2]

Here's some Visual Basic code that demonstrates how to access this C# indexer.

```
' Construct an instance of the BitArray type.
Dim ba as New BitArray(10)

' Visual Basic uses () instead of [] to specify array elements.
Console.WriteLine(ba(2))       ' Displays True or False

' Visual Basic also allows you to access the indexer by its name.
Console.WriteLine(ba.Bit(2))   ' Displays same as previous line
```

In C#, a single type can define multiple indexers as long as the indexers all take different parameter sets. In other programming languages, the IndexerName attribute allows you to define multiple indexers with the same signature because each can have a different name. The reason C# won't allow you to do this is because its syntax doesn't refer to the indexer by name; the compiler wouldn't know which indexer you were referring to. Attempting to compile the following C# source code causes the

[2] For this reason, the IndexerNameAttribute class is not part of the ECMA standardization of the CLI and the C# language.

compiler to generate the following message: `error C0111: Type 'SomeType' already defines a member called 'this' with the same parameter types.`

```
using System;
using System.Runtime.CompilerServices;

public sealed class SomeType {

   // Define a get_Item accessor method.
   public Int32 this[Boolean b] {
      get { return 0; }
   }

   // Define a get_Jeff accessor method.
   [IndexerName("Jeff")]
   public String this[Boolean b] {
      get { return null; }
   }
}
```

You can clearly see that C# thinks of indexers as a way to overload the `[]` operator, and this operator can't be used to disambiguate parameterful properties with different method names and identical parameter sets.

By the way, the `System.String` type is an example of a type that changed the name of its indexer. The name of `String`'s indexer is `Chars` instead of `Item`. This read-only property allows you to get an individual character within a string. For programming languages that don't use `[]` operator syntax to access this property, `Chars` was decided to be a more meaningful name.

Selecting the Primary Parameterful Property

C#'s limitations with respect to indexers brings up the following two questions:

- What if a type is defined in a programming language that does allow the developer to define several parameterful properties?
- How can this type be consumed from C#?

The answer to both questions is that a type must select one of the parameterful property names to be the default property by applying an instance of `System.Reflection.DefaultMemberAttribute` to the class itself. For the record, `DefaultMemberAttribute` can be applied to a class, a structure, or an interface. In C#, when you compile a type that defines a parameterful property, the compiler automatically applies an instance of `DefaultMember` attribute to the defining type and takes it into account when you use the `IndexerName` attribute. This attribute's constructor specifies the name that is to be used for the type's default parameterful property.

So, in C#, if you define a type that has a parameterful property and you don't specify the `IndexerName` attribute, the defining type will have a `DefaultMember` attribute indicating `Item`. If you apply the `IndexerName` attribute to a parameterful property, the defining type will have a `DefaultMember` attribute indicating the string name specified in the `IndexerName` attribute. Remember, C# won't compile the code if it contains parameterful properties with different names.

For a language that supports several parameterful properties, one of the property method names must be selected and identified by the type's `DefaultMember` attribute. This is the only parameterful property that C# will be able to access.

When the C# compiler sees code that is trying to get or set an indexer, the compiler actually emits a call to one of these methods. Some programming languages might not support parameterful properties. To access a parameterful property from one of these languages, you must call the desired accessor method explicitly. To the CLR, there's no difference between parameterless properties and parameterful properties, so you use the same `System.Reflection.PropertyInfo` class to find the association between a parameterful property and its accessor methods.

The Performance of Calling Property Accessor Methods

For simple `get` and `set` accessor methods, the just-in-time (JIT) compiler *inlines* the code so that there's no run-time performance hit as a result of using properties rather than fields. Inlining is when the code for a method (or accessor method, in this case) is compiled directly in the method that is making the call. This removes the overhead associated with making a call at run time at the expense of making the compiled method's code bigger. Because property accessor methods typically contain very little code, inlining them can make the native code smaller and can make it execute faster.

Note The JIT compiler does not inline property methods when debugging code because inlined code is harder to debug. This means that the performance of accessing a property can be fast in a release build and slow in a debug build. Field access is fast in both debug and release builds.