Applied .NET Attributes

TOM BARNABY AND JASON BOCK

Apress™

Applied .NET Attributes
Copyright ©2003 by Tom Barnaby and Jason Bock

ISBN (pbk): 1-59059-136-4

Printed and bound in the United States of America 10987654321

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

Attribute Fundamentals

In the course of developing an application, it is quite typical to have core functionality contained in methods that are invoked by other specialized methods. This reduces the spread of code and improves the maintainability of the code base. Similarly, it is routine to move commonly used data to other generalized levels. However, sometimes it becomes apparent that the shared implementations or the data should be published to a level where the implementations or data can be used across other class definitions. This is where metadata comes into play, as object-related services consume this information to provide reusable implementations.

In this chapter, you'll get a tour of the fundamentals of attributes. You'll see how code is dissected to determine when attributes should be used and when other techniques are more applicable. Then you'll be provided with the essentials of attribute-based programming in .NET. You'll understand how attributes are defined and where they can be applied in an assembly. Finally, you'll get a detailed look at where the attribute's information is stored in an assembly.

Applications of Metadata

Attributes are just like any other tool in the developer's proverbial toolkit. They are useful in some situations; in other cases, they can make the solution much more complex than it needs to be. In the following sections, we'll describe how data is used within code and when data is used to define and describe the code base itself.

Defining Data in the Code

Let's start our journey into metadata by looking at some rather simplistic models of a country. The intent is not to have a class that completely describes a country, but to focus on what the code does. Listing 1-1 shows an initial attempt at coding a class that defines a country. As you'll see in a moment, it's not the finest example of writing code.

Listing 1-1. Defining the BadCountry Class

```
namespace Apress.NetAttributes
{
    public class BadCountry
    {
        public string mName;
        public long mPopulation;

        public BadCountry() : base() {}

        public long Population
        {
            get
            {
                return this.mPopulation;
            }
            set
            {
                if(value < 0 ||
                    value > 5000000000000)
                {
                    throw new ArgumentOutOfRangeException(
                        "value", value,
                        "The given value is out of range.");
                }

                this.mPopulation = value;
            }
        }

        public string Name
        {
            get
            {
                return this.mName;
            }
            set
            {
                this.mName = value;
            }
        }
    }
}
```

Seasoned developers can probably find a number of problems with this implementation. However, the key points of contention with this code as it relates to data quality are as follows:

- The `mName` field is never initialized. If you create an instance of the `BadCountry` class and get the `Name` property, you will have a null reference, which may cause a `NullReferenceException`.

- Whenever the `Population` property is set, the given value is checked to ensure it is within a predefined range. However, this range is parameterized via hard-coded values (zero and five trillion inclusive). If you wanted to change the range's boundary conditions, you would need to find these values within each occurrence of the range check. Furthermore, there is no way for a `BadCountry` client to know that the range exists without resorting to reverse-engineering techniques.

- The exception message is hard-coded. If you wanted to change the message, you would need to do this within the property setter itself. Also, there is no way to reuse the message string in another section of code.

- The fields are declared as public, so any data validation that exists (such as the `Population` setter) can be circumvented with ease.

To address these issues, Listing 1-2 shows a second attempt at constructing a country definition.

Listing 1-2. Improving the BadCountry Definition

```
namespace Apress.NetAttributes
{
    public class ImprovedCountry
    {
        private const string ERROR_ARGUMENT_NAME = "name";
        private const string ERROR_ARGUMENT_POPULATION = "population";
        private const string ERROR_MESSAGE_NULL_VALUE =
            "The given value should not be null.";
        private const string ERROR_MESSAGE_OUT_OF_RANGE =
            "The given value is out of range.";
        public const long MINIMUM_POPULATION = 0;
        public const long MAXIMUM_POPULATION = 5000000000000;

        protected string mName = string.Empty;
        protected long mPopulation;
```

```
private ImprovedCountry() : base() {}

public ImprovedCountry(string name, long population)
{
    this.CheckInvariantName(name);
    this.CheckInvariantPopulation(population);

    this.mName = name;
    this.mPopulation = population;
}

protected void CheckInvariantName(string name)
{
    if(name == null)
    {
        throw new ArgumentNullException(
            ImprovedCountry.ERROR_ARGUMENT_NAME,
            ImprovedCountry.ERROR_MESSAGE_NULL_VALUE);
    }
}

protected void CheckInvariantPopulation(long population)
{
    if(population < ImprovedCountry.MINIMUM_POPULATION ||
        population > ImprovedCountry.MAXIMUM_POPULATION)
    {
        throw new ArgumentOutOfRangeException(
            ImprovedCountry.ERROR_ARGUMENT_POPULATION, population,
            ImprovedCountry.ERROR_MESSAGE_OUT_OF_RANGE);
    }
}

public long Population
{
    get
    {
        return this.mPopulation;
    }
    set
    {
        this.CheckInvariantPopulation(value);
        this.mPopulation = value;
    }
}
```

```
        public string Name
        {
            get
            {
                return this.mName;
            }
            set
            {
                this.CheckInvariantName(value);
                this.mName = value;
            }
        }
    }
}
```

At first glance, it looks like there is more code involved, and that is true. There are 39 lines in the `BadCountry` definition and 72 lines in the `ImprovedCountry` definition. However, the code base has been vastly improved. The fields are now `protected`, so any changes to their values must be done through the Application Programming Interface (API) set up by `ImprovedCountry`[1] or via inheritance.

Furthermore, any time the field values need to be changed, validation rules are enforced via calls to either `CheckInvariantName()` or `CheckInvariantPopulation()` in the constructor and the setters of both properties. The error messages are now stored in constants and are available for reuse in future version of `ImprovedCountry`. Finally, the minimum and maximum population values are stored as public constants, so clients will know what the range is before they write even one line of code against an `ImprovedCountry` instance.

All of these improvements in the code are fairly standard in any refactoring phase. However, these changes don't necessarily make the code bulletproof. We think it's a good idea to move so-called *magic* values (values that just pop up in code) into constants. But the problem with constants is that they are not automatically updated in client code. For example, let's say there was a WinForm application that used the `ImprovedCountry` class, and it showed the minimum and maximum population values to the user, like this:

```
namespace Apress.NetAttributes
{
    public class CountryClient : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Label lblMaximumPopulationValue;
        private System.Windows.Forms.Label lblMinimumPopulationValue;
```

1. It is possible to change the value of a private field via the Reflection API, but we'll ignore that issue for now.

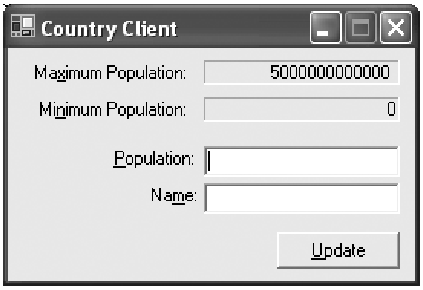
```
        public CountryClient()
        {
            InitializeComponent();
            this.ShowCountry();
            this.ShowPopulationLimits();
        }

        private void ShowPopulationLimits()
        {
            this.lblMaximumPopulationValue.Text =
                ImprovedCountry.MAXIMUM_POPULATION.ToString();
            this.lblMinimumPopulationValue.Text =
                ImprovedCountry.MINIMUM_POPULATION.ToString();
        }

        static void Main()
        {
            Application.Run(new CountryClient());
        }

        //  ...
    }
}
```

Let's say that population increases and improvements in space exploration now require us to change the maximum population.[2]

```
public const long MAXIMUM_POPULATION = 10000000000000;
```

We now take version 2.0.0.0 of the `Country` assembly and put it into the `CountryClient`'s application directory. Figure 1-1 shows the results of running the application.

2. We're assuming that .NET will be around long enough to be used when the colonization of space becomes commonplace.


Country Client

Maximum Population: 5000000000000

Minimum Population: 0

Population:

Name:

Figure 1-1. Displaying constant values in an updated assembly

The results may surprise you at first glance. Why didn't the client application pick up the new boundary value? The reason is that constant values are embedded into a client assembly at compile-time. Take a look at the resulting Common Intermediate Language (CIL) code[3] that is in `ShowPopulationValues()`.

```
.method private hidebysig instance
  void ShowPopulationLimits() cil managed
{
  .maxstack  2
  .locals init (int64 V_0)
  IL_0000:  ldarg.0
  IL_0001:  ldfld class [System.Windows.Forms]System.Windows.Forms.Label
    Apress.NetAttributes.CountryClient::lblMaximumPopulationValue
  IL_0006:  ldc.i8 0x48c27395000
  IL_000f:  stloc.0
  IL_0010:  ldloca.s V_0
  IL_0012:  call instance string [mscorlib]System.Int64::ToString()
  IL_0017:  callvirt instance void [System.Windows.Forms]
    System.Windows.Forms.Control::set_Text(string)
  IL_001c:  ldarg.0
  IL_001d:  ldfld class [System.Windows.Forms]System.Windows.Forms.Label
    Apress.NetAttributes.CountryClient::lblMinimumPopulationValue
  IL_0022:  ldc.i4.0
  IL_0023:  conv.i8
  IL_0024:  stloc.0
  IL_0025:  ldloca.s V_0
  IL_0027:  call instance string [mscorlib]System.Int64::ToString()
  IL_002c:  callvirt instance void [System.Windows.Forms]
    System.Windows.Forms.Control::set_Text(string)
  IL_0031:  ret
}
```

3. If you're not familiar with CIL, one of the authors of this book, Jason Bock, has written a book on this base language of .NET: *CIL Programming: Under the Hood of .NET* (Apress, 2002).

The lines that are of particular importance are `IL_0006` and `IL_0022`. The value, 0x48c27395000, is 5,000,000,000,000, and `ldc.i4.0` loads 0 onto the stack. Essentially, the constant values are hard-coded into the client's code base, so any updates go unnoticed.[4]

Again, constants are not necessarily a bad thing. But creating public constants does not mean that clients will ever use those constants in the way that the implementer intended. Naming a field `MAXIMUM_POPULATION` has some human-based semantics,[5] but it's up to a client to use that field effectively. Moreover, as you have just seen, constants do not have a good versioning story. Sometimes, it's advantageous to have data that can be versioned and is accessible to a client. As you'll see in the "Attributes in .NET" section later in this chapter, attributes have these properties.

Defining Data About the Code

As with anything that is associated with data, there comes a time when an object's information will need to be persisted so it can be retrieved (and possibly updated) in the future. One approach is to have a client determine how the object's state should be saved. Listing 1-3 demonstrates one possible implementation.

Listing 1-3. Saving a Country Object

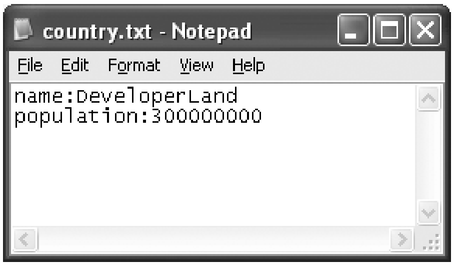
```
private void btnSave_Click(object sender, System.EventArgs e)
{
    if(this.mCountry != null)
    {
        TextWriter countryFile =
            File.CreateText(Application.StartupPath +
            @"\country.txt");

        try
        {
            countryFile.WriteLine(
                "name:{0}", this.mCountry.Name);
            countryFile.WriteLine(
                "population:{0}", this.mCountry.Population);
        }
        finally
```

4. This is done for performance reasons. It's a lot faster to simply load a value than it is to read it from a field in either a class or an object.

5. This is especially true for those who can read English.

```
        {
            countryFile.Close();
        }
    }
}
```

Figure 1-2 shows what a typical country.txt file would look like in Notepad.



```
name:DeveloperLand
population:300000000
```

Figure 1-2. Persisting an ImprovedCountry object

However, this implementation is just one way to save the data to a file. What if another client application decides to save it to an XML file? What if the second client chooses to store the `Population`'s property value first? Persistence strategies can vary widely among applications. The only chance of interoperability between these formats is solid documentation.

One way to improve this approach is to create an interface that a class can implement so it can control the serialization process:

```
public interface IPersist
{
    void Load(Stream persistenceTarget);
    void Save(Stream persistenceTarget);
}
```

To illustrate how this interface could be implemented, we'll create a new class named `PersistedCountry` that has `ImprovedCountry` as its base class. It will also implement the `IPersist` interface. Listing 1-4 shows the implementation of `PersistedCountry`.

Listing 1-4. Implementing PersistedCountry

```
public class PersistedCountry : ImprovedCountry, IPersist
{
    private const string NAME_KEY = "name";
    private const string POPULATION_KEY = "population";
    public PersistedCountry(string name, long population) :
        base(name, population) {}

    public void Load(Stream persistenceTarget)
    {
        int totalBytes = (int)persistenceTarget.Length;
        Decoder dec = (new UnicodeEncoding()).GetDecoder();
        byte[] storedInfo = new byte[totalBytes];
        persistenceTarget.Read(storedInfo, 0, totalBytes);
        char[] storedChars = new char[dec.GetCharCount(
            storedInfo, 0, totalBytes)];
        int totalDecodedChars = dec.GetChars(storedInfo, 0, totalBytes,
            storedChars, 0);
        string info = new string(storedChars);

        int nameStart = info.IndexOf(NAME_KEY);
        int populationStart = info.IndexOf(POPULATION_KEY);

        this.mName = info.Substring(nameStart + NAME_KEY.Length + 1,
            populationStart - (nameStart + NAME_KEY.Length));
        this.mPopulation = Int32.Parse(
            info.Substring(populationStart + POPULATION_KEY.Length + 1));
    }

    public void Save(Stream persistenceTarget)
    {
        Encoding enc = new UnicodeEncoding();
        string nameInfo = string.Format("{0}:{1}", NAME_KEY, this.mName);
        byte[] nameInfoBytes = enc.GetBytes(nameInfo);
        persistenceTarget.Write(nameInfoBytes, 0,
            nameInfoBytes.Length);
        string populationInfo = string.Format("{0}:{1}",
            POPULATION_KEY, this.Population);
        byte[] populationInfoBytes = enc.GetBytes(populationInfo);
        persistenceTarget.Write(populationInfoBytes, 0,
            populationInfoBytes.Length);
    }
}
```

This will centralize how the `ImprovedCountry` object will save its information, but there is still the problem of consistency. One class may decide to use XML; another might use a binary format. Furthermore, there is no chance to share these persistence mechanisms among class definitions as long as they remain internal to the object.

If we alter IPersist's methods to take an object that stores name/value pairs instead of a Stream-based object, the object is responsible only for storing the data that it wants to retain. To illustrate this, we'll overload Load() and Save() to take a Hashtable reference.

```
public interface IPersist
{
    void Load(Stream persistenceTarget);
    void Load(Hashtable nameValuePairs);
    void Save(Stream persistenceTarget);
    void Save(Hashtable nameValuePairs);
}
```

This reduces the burden of the implementer of IPersist significantly.

```
public void Load(Hashtable nameValuePairs)
{
    if(nameValuePairs != null)
    {
        if(nameValuePairs.Contains(NAME_KEY))
        {
            this.mName = (string)nameValuePairs[NAME_KEY];
        }
        if(nameValuePairs.Contains(POPULATION_KEY))
        {
            this.mPopulation = (long)nameValuePairs[POPULATION_KEY];
        }
    }
}

public void Save(Hashtable nameValuePairs)
{
    if(nameValuePairs != null)
    {
        nameValuePairs.Add(NAME_KEY, this.mName);
        nameValuePairs.Add(POPULATION_KEY, this.mPopulation);
    }
}
```

The client would now persist an ImprovedCountry object by using another object to house the storage-strategy code.

```
public interface IObjectStorage
{
    void Load(IPersist targetObject, FileStream stream);
    void Save(IPersist targetObject, FileStream stream);
}
```

Now the serialization apparatus is generalized, not specific to how one object decides to do it.

NOTE *At this point, the design is starting to take on the trappings of the serialization classes in .NET. It's nowhere near as comprehensive, but the point of this discussion is not to repeat the designs of the .NET Framework, so we'll stop here.*

We can take this one step further. Most of the time, an object will store its field values during a serialization process. Via Reflection, code can be written to read an object's fields (public, private, or otherwise) in a general fashion that is independent of the object's type.[6]

```
public void PokeAtObject(object targetObject)
{
    FieldInfo[] fields = targetObject.GetType().GetFields(
        BindingFlags.Public | BindingFlags.NonPublic |
        BindingFlags.Instance);

    foreach(FieldInfo field in fields)
    {
        string fieldName = field.Name;
        object fieldValue = field.GetValue(targetObject);
    }
}
```

In this case, there is no reason for an object to implement `IPersist`, because it does not need to customize the serialization process. Granted, object persistence is not as easy as this. For example, the Reflection code doesn't take into consideration deep object graphs; it just looks at the fields contained within a given object. Furthermore, some objects may have specialized needs that simple field iteration won't handle, so having the `IPersist` interface is a good thing. It should, however, not be a requirement for a class that wants its instances to be

6. Such access requires high privileges, which is accounted for in .NET's security architecture.

persistable to implement an interface. This is where using attributes can be beneficial. Attributes are useful to define public data about an assembly's members that can be used by object-related services to perform generalized tasks.

Another example of where attributes are useful is when you need to mark either methods or entire classes as obsolete. If it were just one method, a developer could have the method throw a `NotSupportedException`, but clients of that object would get an unexpected surprise when they called that method. Also, there is no way to warn the client of future obsolescence. An attribute could be used on a method or an entire class to state that other paths should be used in the near future. Plus, an attribute can be constructed to provide fair warning without causing unexpected behavior.

Defining Data Outside the Code

There is one last place where we can put data that defines our code: in a configuration file. This is nothing new. Long-time Windows developers have used INI files to store application-related information like user preferences, FTP site locations, and so on. Now it's conceivable that you could store metadata in configuration files. For example, here's what a configuration file would look like if we needed to state that `BadCountry` was obsolete:

```
<BadCountry>
    <Metadata>
        <Obsolete CausesError="True"/>
    </Metadata>
</BadCountry>
```

The runtime would need to look up the information in the configuration file whenever a serialization request was made to make sure that the designer of the class allows its instances to be persisted.

However, there are problems with this approach. First, with the metadata outside the code base, you now have an installation issue. Having the metadata in an XML file means that you must ship two files to the client. When the metadata is in the assembly, you need to ship only one file. This may not sound like a big deal, but at least with the one-file approach (assembly-only), you know the metadata is there when the assembly is there.

Another problem is with metadata inheritance. As you'll see in the "Attribute Targets" section later in this chapter, as well as the "Inheritance and Custom Attributes" section in Chapter 4, you can set up attributes so that their use on a base class will also affect subclasses. You could do this with configuration files as well, but this kind of behavior is not designed into .NET. Code that acts on metadata would need to consider the lookup of the information within a base class; this is automatically taken care of through the design of attributes in .NET.

The bigger issue is that external metadata can be adjusted by a client with relative ease. Would you want someone to make a class that was obsolete relevant again? What about making a class serializable when it was never intended to be saved to disk? When the metadata is in the assembly, it's much harder for someone to change that metadata. The intentions of the designer have a better chance of being preserved when they're embedded within the compiler's output.

NOTE *Granted, we're not dealing with reverse-engineering scenarios here. Someone who is savvy enough with ILDasm and CIL could make an obsolete class nonobsolete rather easily. There are countermeasures to reverse-engineering, but that discussion goes well beyond the topic at hand.*

Don't get us wrong—configuration files can be useful, especially when they define values that are used by the code and may change after the code is compiled. But configuration files are not appropriate when they define design characteristics of the class itself. Attributes are tightly coupled to the element that they are associated with, which is an essential feature of metadata.

You've now seen where attributes can be used effectively. Now it's time to see how attributes are used in C#.

Attributes in .NET

Using attributes in your .NET code is fairly straightforward, but knowing the rules will not only help you to understand why some compilation errors may occur when you code with attributes, but it will also guide you when you create your own attributes (a topic we'll cover in Chapter 4). In this section, you'll learn how you declare attributes in your C# code, where and when attributes can be declared, and where the attribute information ends up.

Declaration Fundamentals

To demonstrate how attributes are used in C# code, we'll use the `ObsoleteAttribute` class to mark our `BadCountry` class as a class that clients should no longer use, but it will not cause an error to use `BadCountry`. Listing 1-5 shows the code necessary to do this.

Listing 1-5. Making a Class Obsolete via Attributes

```
[Obsolete]
public class BadCountry
{
    //  ...
}
```

Attributes are declared within the brackets. Only objects whose class inherits from System.Attribute can be used within the brackets. For example, the following code would cause an error:

```
[DateTime(2003, 2, 19)]
public class BadCountry
{
    //  ...
}
```

Attributes are usually named with the string "Attribute" at the end. You are not required to type in the full name when you add the attribute to your code,[7] but you can if you prefer, as shown here:

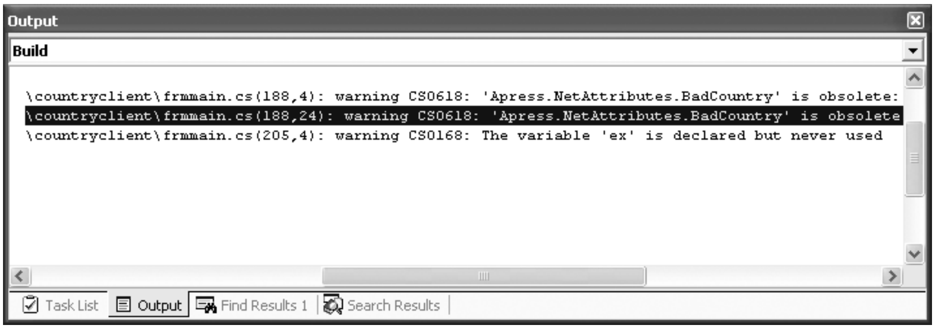
```
[ObsoleteAttribute]
public class BadCountry
{
    //  ...
}
```

When you add an attribute in your code, you are technically creating a new object whose information will be stored in the assembly. If the attribute has a no-argument constructor, you don't need to use parentheses in your declaration. However, as is the case with ObsoleteAttribute, attributes can define custom constructors so you can set the state of the attribute with more detail. The following code snippet uses a custom constructor of ObsoleteAttribute to set a string and boolean value.

7. This is primarily done to reduce name collisions between attribute and normal class definitions within an assembly.

```
[Obsolete("This class should no longer be used - switch to ImprovedCountry.",
    false)]
public class BadCountry
{
    //  ...
}
```

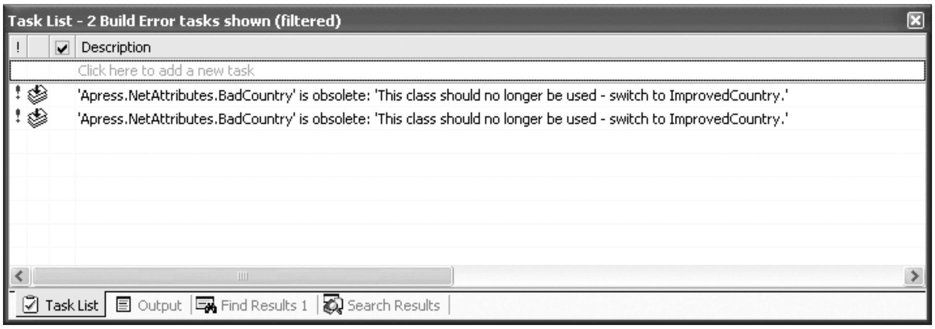
In this case, the `string` argument (called `message`) is used to define a descriptive message that clients can use to find other alternative implementations. The `boolean` argument (called `error`) is used during the client's compilation process. A `true` value will cause a compilation error if the deprecated item is still used; a `false` value will cause a compilation warning. Figure 1-3 shows the compilation output in Visual Studio .NET when `BadCountry` is used with `error` set to `false`.



```
\countryclient\frmmain.cs(188,4): warning CS0618: 'Apress.NetAttributes.BadCountry' is obsolete:
\countryclient\frmmain.cs(188,24): warning CS0618: 'Apress.NetAttributes.BadCountry' is obsolete
\countryclient\frmmain.cs(205,4): warning CS0168: The variable 'ex' is declared but never used
```

Figure 1-3. Compilation warning messages when using ObsoleteAttribute

However, as Figure 1-4 demonstrates, when `error` is set to `true`, the client code won't even compile.


Task List - 2 Build Error tasks shown (filtered)
!
Description
Click here to add a new task
Task List
Output
Find Results 1
Search Results

```
'Apress.NetAttributes.BadCountry' is obsolete: 'This class should no longer be used - switch to ImprovedCountry.'
'Apress.NetAttributes.BadCountry' is obsolete: 'This class should no longer be used - switch to ImprovedCountry.'
```

Figure 1-4. Compilation error messages when using ObsoleteAttribute

Not all attributes will affect the compilation process as `ObsoleteAttribute` does. In fact, attributes in themselves are harmless, *unless* some other process, like a compiler or the Common Language Runtime (CLR), acts on them. Although attributes will end up in your assembly, they cannot affect your code.[8]

NOTE *Other factors are involved in using an attribute's information and acting accordingly. Chapters 2 and 3 will cover different attributes and when they come into play.*

Attribute Targets

So far, you've seen how an attribute is declared in C# code to affect a class definition. However, attributes can be applied to nearly any part of a .NET assembly. When you declare an attribute as shown in Listing 1-5, the attribute is applied to the class, rather than to any of the class's methods or fields. If you want to make the target of an attribute explicit, you can prefix the attribute declaration with the target name by adding the `type` keyword, as Listing 1-6 shows.

Listing 1-6. Explicitly Stating the Attribute Target

```
[type: Obsolete(
    "This class should no longer be used - switch to ImprovedCountry.",
    true)]
public class BadCountry
{
    // ...
}
```

When an attribute is defined, it can also state the valid targets that it can be applied to in code. This is done via the `AttributeTargets` enumeration. Table 1-1 describes all of the values in the `AttributeTargets` enumeration.

8. Of course, this is not true if a compiler is designed to read attributes to emit code during the compilation process.

Table 1-1. Valid Attribute Targets

AttributeTargets Name	Description
All	All application elements
Assembly	Assembly level
Class	All classes
Constructor	All constructors
Delegate	All delegates
Enum	All enumerations
Event	All events
Field	All fields
Interface	All interfaces
Method	All methods
Module	All modules
Parameter	All method arguments
Property	All properties
ReturnValue	All method return values
Struct	All value types

The values of `AttributeTargets` can be ORed together, so an attribute designer can make any combination of valid targets. The `All` value is supplied if an attribute can be applied to any target. C# defines nine keywords that can be used to apply the attribute to a specific target in code: `assembly`, `event`, `field`, `method`, `module`, `param`, `property`, `return`, and `type`. As you can guess, some of these keywords overlap with the values in `AttributeTargets`. For example, `type` can be used on a class, an interface, a structure, an enumeration, and a delegate definition; `param` can be used on only method arguments.

Where an attribute can be legally applied is determined by the implementer of the attribute. An attribute designer uses the `AttributeTargetsAttribute` class on the attribute definition itself to control the attribute's valid destinations. (We'll cover `AttributeTargetsAttribute` in detail in Chapter 4.) For example, if you need to make only one method of `BadCountry` obsolete, you apply the attribute like this:

```
[property: Obsolete("This is a bad property.",
    true)]
public long Population
{
    // ...
}
```

However, you cannot make an entire assembly obsolete. The following C# code will cause a compilation error:[9]

```
[assembly: Obsolete(
    "This assembly should no longer be used.",
    true)]
```

Furthermore, the location of the attribute when a target is explicitly given is also important. For example, an assembly-level attribute (like `AssemblyTitleAttribute`) cannot be declared inside a class or namespace; it must exist outside these scopes, as the following code snippet demonstrates:

```
//  This is OK.
[assembly: AssemblyTitle("Country assembly.")]

namespace Apress.NetAttributes
{
    //  This isn't.
    [assembly: AssemblyTitle("Country assembly.")]

    public class BadCountry
    {
        //  Neither is this.
        [assembly: AssemblyTitle("Country assembly.")]
    }
}
```

You're not limited to applying only one attribute to a given target. For example, the following code is valid:

9. The .NET Framework SDK will list in an attribute's definition which targets are valid.

```
[type: Obsolete(
    "This class should no longer be used - switch to ImprovedCountry.",
    true)]
[type: Serializable]
public class BadCountry
{
    // ...
}
```

You may, however, be limited in terms of how many times you can apply one *specific* attribute to a particular target. For example, the following code will not compile:

```
[type: Obsolete("This class should no longer be used - switch to ImprovedCountry.",
    true)]
[type: Obsolete("Really - don't use this class!",
    true)]
public class BadCountry
{
    // ...
}
```

As with the valid target locations, an attribute designer can control if an attribute can be applied to same target multiple times via the `AllowMultiple` property of the `AttributeUsageAttributes` class. The default behavior for an attribute is that it will not be a multiuse attribute, because it is unusual for an attribute to be applied to the same target multiple times, but this can be changed.

Finally, attributes can control whether or not their information is inherited in subclasses or overridden methods. For example, let's say we tried to use `BadCountry` as a base class.

```
[type: Obsolete("Really - don't use this class!",
    true)]
public class BadCountry
{
    // ...
}

//  This will not compile.
public class BadInheritedCountry : BadCountry
{
    // ...
}
```

In this case, the code won't compile because `BadInheritedCountry` is trying to use an obsolete class. However, if we didn't set `error` to `true` in `ObsoleteAttribute`'s constructor, not only would the code compile, but clients would be able to do this without getting a warning.

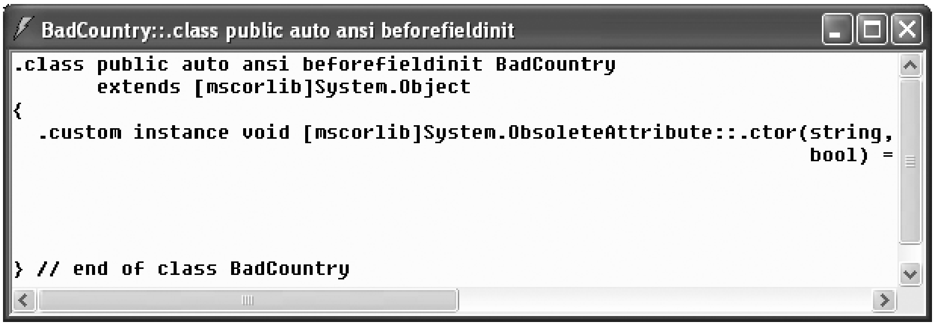
```
BadInheritedCountry bic = new BadInheritedCountry();
```

`ObsoleteAttribute` was designed so that its information does not flow to a subclass. However, this choice is up to the creator of the attribute. Again, `AttributeUsageAttributes` is the center of control to determine how an attribute's information flows in inheritance scenarios. The default behavior is for attributes to be inheritable,[10] but an attribute can be limited to the target to which it is applied. We'll come back to these design issues in the "Inheritance and Custom Attributes" section in Chapter 4.

You now know how attributes work in C#. But the story doesn't stop there. When you compile your attribute-laden code, the metadata ends up in the assembly in one form or another. In the next section, you'll see where the attribute is located in an assembly and how that information is stored.

Compiling Attributes

Fortunately, from a compilation perspective, there is nothing that you need to do differently when you include attributes in your C# code. You don't need to check a check box in Visual Studio .NET or add a switch to csc.exe to have your attributes show up in the resulting assembly. For example, if you look at an assembly that contains attributes in ILDasm, you'll see something that looks like Figure 1-5.



```
.class public auto ansi beforefieldinit BadCountry
       extends [mscorlib]System.Object
{
  .custom instance void [mscorlib]System.ObsoleteAttribute::.ctor(string,
                                                                   bool) =



} // end of class BadCountry
```

Figure 1-5. The .custom directives in an assembly

10. This has nothing to do with whether or not the attribute *class* is sealed.

The custom attribute will be stored within a `.custom` directive. The location of the directive will vary depending on the target stated in the C# code. In this case, `ObsoleteAttribute` was used on the `BadCountry` class. The directive will state the full name of the attribute type, along with the constructor used in the attribute declaration. Unfortunately, space limitations prevented us from showing the full story in Figure 1.5. There is something after the equal sign that couldn't fit in the image. Listing 1-7 shows what ends up on the other side.

Listing 1-7. The Full Attribute Format in an Assembly

```
.class public auto ansi beforefieldinit BadCountry
  extends [mscorlib]System.Object
{
  .custom instance void
    [mscorlib]System.ObsoleteAttribute::.ctor(string,
    bool) =
    ( 01 00 40 54 68 69 73 20 63 6C 61 73 73 20 73 68   // ..@This class sh
     6F 75 6C 64 20 6E 6F 20 6C 6F 6E 67 65 72 20 62    // ould no longer b
     65 20 75 73 65 64 20 2D 20 73 77 69 74 63 68 20    // e used - switch
     74 6F 20 49 6D 70 72 6F 76 65 64 43 6F 75 6E 74    // to ImprovedCount
      72 79 2E 01 00 00 )                              // ry....
} // end of class BadCountry
```

You can glean some general ideas as to what the bytes represent, but you do not need to know the full details of the format to use and create attributes successfully. As you'll see in Chapter 4, the API to read metadata from an assembly is rather straightforward and does not require you to know the layout of the byte array.

NOTE *For those who are curious to know what the bytes stand for, Appendix of this book goes through the format in excruciating detail.*

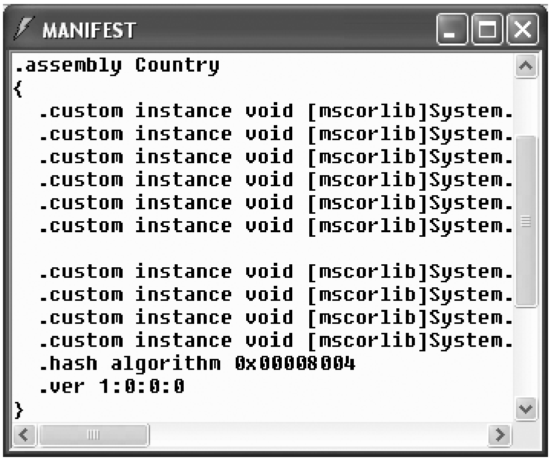
Most attributes will end up in a `.custom` directive. Interestingly enough, they are known as *custom* attributes. However, there is a special set of attributes that, when present in code, will end up in other places in the assembly. These are known as *pseudo-custom* attributes.[11] You declare them in the same way that you

11. See Section 20.2.1 of Partition II for more details on pseudo-custom attributes. Note that some pseudo-custom attributes are not CLS-compliant, so take care if you use them in your code.

declare any other attribute in your C# code, but the end results in the assembly are vastly different. For example, take a look at the following code:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

When you compile your code, you won't find a `.custom` directive with a type name that contains `AssemblyVersionAttribute`. Figure 1-6 shows what happens when you use `AssemblyVersionAttribute`.



```
.assembly Country
{
  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.

  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.
  .custom instance void [mscorlib]System.
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
```

Figure 1-6. Pseudo-custom attributes in assemblies

In this case, the `.ver` directive contains the version information of the assembly.

There is a reason why pseudo-custom attributes are stored in this fashion: it's more efficient. Pseudo-custom attributes don't have the overhead of storing a bunch of bytes that a normal custom attribute requires. An explicit example of this is when you make a class serializable, like this:

```
[type: Serializable]
public class SerializableCountry
{
    // ...
}
```

The ILDasm results are as follows:

```
.class public auto ansi serializable beforefieldinit SerializableCountry
  extends [mscorlib]System.Object
{
} // end of class SerializableCountry
```

Notice that the `serializable` attribute now ends up in the class definition. There is no `.custom` directive anywhere on the class. What is really going on is that a bit flag is being set for `SerializableCountry`. When you use the `SerializableAttribute` attribute with the class, here's what the token looks like:[12]

```
TypDefName: Apress.NetAttributes.SerializableCountry  (02000008)
Flags     : [Public] [AutoLayout] [Class] [Serializable] [AnsiClass]  (00102001)
```

Without the `SerializableAttribute` attribute, you'll notice that a value in the `Flags` field changes:

```
TypDefName: Apress.NetAttributes.SerializableCountry  (02000008)
Flags     : [Public] [AutoLayout] [Class] [AnsiClass]  (00100001)
```

Pseudo-custom attributes have this special storage consideration because they are used extensively by compilers or by the CLR. It's much faster for the CLR to look at a bit field to see if it's serializable, rather than to go through all of the custom attributes (if any exist) to see if one of them is of the `SerializableAttribute` type. Of course, with this efficiency comes a trade-off in verbosity. A class is either serializable or it's not serializable. Contrast that with the `ObsoleteAttribute`, which lets you give a client a helpful message and make obsolete elements cause either a warning or an error during compilation.

SOURCE CODE *The code for these examples is in Chapter1\Country and Chapter1\CountryClient.*

12. You can see these flag values (along with a lot more juicy metadata information) if you start up ILDasm with the /adv switch. A Metadata option will appear on the View menu, which you can use to view the token and flag values.

Conclusion

In this chapter, you learned the following about attributes:

- Where data can be stored in and around code, and how that compares to attributes
- How to declare attributes within C# code
- Where attributes end up in an assembly

In the next chapter, you'll get the details on how some attributes are used directly during the compilation process and the effects of using these attributes.