Custom Attributes

Source http://articles.techrepublic.com.com/5100-3513-6175490.html

Attributes are special classes that can be applied to classes, properties, and methods at design time. Attributes provide a way to describe certain aspects of an element or determine the behavior of other classes acting upon the element. Those descriptions and behaviors can then be accessed and examined at runtime. You can think of attributes as a way of adding special modifiers to your class members.

For example, if you have written Web services, you are no doubt aware that the `WebMethod` attribute must be applied to methods for them to be exposed through the service. This is a perfect example to show the usage of attributes because the `WebMethod` attribute is used to extend the programming model. There is no built-in way in C# of signifying that a method should be exposed through the Web service (as there is, for example, of signifying that a method should be private), so the `WebMethod` attribute was written to satisfy this need.

Developing custom attributes

The process of creating a custom attribute is very simple. There are just a few things you must take into account before creating the attribute:

- **What is the purpose of the attribute?**
  Attributes can be used in any number of ways. You need to define what exactly the attribute is meant to accomplish and make sure that specific functionality isn't already covered by built-in .NET Framework assemblies. It is better to use first-class .NET modifiers than attributes, as this simplifies the integration process with other assemblies.
- **What information must the attribute store?**
  Is this attribute intended to be a simple flag to indicate a certain capability or will the attribute have to store information? An attribute can hold a set of information given to it at design time and expose that information at runtime.
- **In which assembly should the attribute reside?**
  In most cases, it is okay to include the attributes in the same assembly that will be using them. However, there are instances when it is better to place the attributes inside of a common, lightweight, shared assembly. This type of configuration allows clients to use the attributes without referencing unneeded assemblies.
- **Which assemblies will recognize the attribute?**
  An attribute isn't worth anything if there are no modules that read it. You will most likely place the classes that read the attribute inside of the same assembly in which the attributes reside. However, as mentioned above, there are instances when you want the logic that reads the attributes, and the attributes themselves, in different assemblies.

Using attributes

Before we get into the details of how to create custom attributes, we need to look at how they are used. For example, assume we have an attribute called "Hide" which effectively

hides properties so that they don't print to the screen. If we were to apply this attribute to the "SSN" property, our code would look like Listing A.

Listing A

```
[Hide()]
publicstring SSN
{
get { return _ssn; }
set { _ssn = value; }
}
```

As a more complicated example, assume we have an attribute called "Alias". This attribute's job is to determine the aliases a property may have. This allows the property's value to be mapped into another property even if the property names don't match. This attribute accepts a series of string values to hold as the mapping names (Listing B).

Listing B

```
[Alias("FirstName", "First")]
publicstring FName
{
get { return _fName; }
set { _fName = value; }
}
```

In this case, the property "FName" is mapped to both "FirstName" and "First." See the example application for more detail on this type of usage.

Creating attributes

Creating attributes is a simple process. You define a class with the data you want to store, and inherit from the `System.Attribute` **class. Listing C is an example of how to create the "Alias" attribute shown in the previous section.**

Listing C

```
classAlias : System.Attribute
{
string[] _names;

public Alias(paramsstring[] names)
{
this.Names = names;
}

publicstring[] Names
{
get { return _names; }
set { _names = value; }
}
}
```

As you can see, this is just a normal class and, with the exception of inheriting from `System.Attribute`, we didn't have to do anything special to enable it to be an attribute. We simply defined the constructor that needed to be used and created a property and private member to store the data.

Listing D is a much simpler attribute -- the "Hide" attribute. This attribute requires no constructor (it uses the default) and doesn't store any data. This is because this attribute is simply a "flag" type attribute:

Listing D

```
classHide : System.Attribute
{
//This is a simple attribute, that only requires
// the default constructor.
}
```

Reading attributes from code

Reading an attribute and examining its data is significantly more complicated than either using an attribute or creating an attribute. Reading an attribute requires the developer to have a basic understanding of how to use reflection on an object. If you are unfamiliar with reflection, you may want to read my "Applied reflection" article series.

Let's assume that we are examining a class and we want to determine which properties have the Alias attribute applied and which aliases are listed. Listing E implements this logic:

Listing E

```
privateDictionary<string, string> GetAliasListing(Type destinationType)
{
//Get all the properties that are in the
// destination type.
PropertyInfo[] destinationProperties = destinationType.GetProperties();
Dictionary<string, string> aliases = newDictionary<string, string>();

foreach (PropertyInfo property in destinationProperties)
{
//Get the alias attributes.
object[] aliasAttributes =
property.GetCustomAttributes(typeof(Alias), true);

//Loop through the alias attributes and
// add them to the dictionary.
foreach (object attribute in aliasAttributes)
foreach (string name in ((Alias)attribute).Names)
aliases.Add(name, property.Name);

//We also need to add the property name
// as an alias.
aliases.Add(property.Name, property.Name);
}
return aliases;
}
```

The most important lines of this section of code are where we call `GetCustomAttributes` **and the section where we loop through the attributes and extract the aliases.**

The `GetCustomAttributes` **method is available from the** `PropertyInfo` **class that we extracted from the object's Type. In the usage shown above, we tell the** `GetCustomAttributes` **method what type of attribute we're looking for and also pass "true" to enable it to pull inherited attributes. The** `GetCustomAttributes` **method returns an object array if any matching attributes are found. There is also another overload of the method that allows you to pull all attributes on the property, regardless of the attribute's type.**

***~~~ End of Article ~~~***