Fun With Attributes

Introduction

The purpose of an `Attribute` is to attach some form of metadata to a property, object, or method, and by using reflection, gain access to said metadata at runtime.

I've extended that paradigm to include functionality that checks whether or not the value of the property is "valid".

Why I Did It

I recently posted an article regarding the parsing of CSV files. I write that code because I must import data from CSV files, and due to the nature of my environment.

I must go to retarded levels to ensure that a human somewhere in the data handling chain didn't somehow screw up the data.

To that end, I have established values for each expected column of data that indicates a problem was encountered while importing the file.

These values are usually "ERROR" for strings, -1 for numeric values, etc, etc. So, I figured, "Hey! I'll take this opportunity to play with custom attributes."

The Code

The InvalidValueAttribute Class

The class declaration is important because that's where you tell the attribute where it can be used, and whether or not to allow multiple instances. Ironically enough, that's done with an attribute, as seen below.

```
// This attribute is only available for properties, and more than one instance of this
// attribute can be specified for a given property. Each instance of attribute must use
// a different trigger type. It is up to the programmer to ensure that Equal and NotEqual
// are not set to the same TriggerValue. To say the results would be unexpected is a huge
// understatement.
[System.AttributeUsage(System.AttributeTargets.Property, AllowMultiple=true)]
public class InvalidValueAttribute : System.Attribute
```

In order to control how the comparison is performed, I defined an enum inside the class.

```
/// <summary>
/// A flag used to indicate how the comparison for validity is performed
/// </summary>
public enum TriggerType
{
    Valid,    // comparison returns TRUE if the property value is != the trigger value
    Equal,    // comparison returns TRUE if the property value is == the trigger value
    NotEqual, // comparison returns TRUE if the property value is != the trigger value
    Over,     // comparison returns TRUE if the property value is <= the trigger value
    Under     // comparison returns TRUE if the property value is >= the trigger value
};
```

Then I implement the configuration properties.

```
/// <summary>
/// Gets/sets a flag indicating how the valid status is determined.
/// </summary>
public TriggerType Trigger       { get; protected set; }

/// <summary>
/// Gets/sets the value that will be used to determine if the property value is valid
/// </summary>
public object      TriggerValue  { get; protected set; }

/// <summary>
/// Gets/sets the expected type that the property value will/should be
/// </summary>
public Type        ExpectedType  { get; protected set; }

/// <summary>
/// Gets/sets the value that was compared against the trigger value so that the
/// TriggerMessage can be constructed.
/// </summary>
public object      PropertyValue { get; protected set; }
```

This property allows the object that contains the decorated property to display the validity status.

```
/// <summary>
/// Gets the trigger message. Called by the method performing the validity check, usually
/// if the value is not valid.
/// </summary>
public string TriggerMsg
{
    get
    {
        string format = string.Empty;
        switch (this.Trigger)
        {
            case TriggerType.Valid    :
            case TriggerType.Equal    : format = "equal to";     break;
            case TriggerType.NotEqual : format = "not equal to"; break;
            case TriggerType.Over     : format = "greater than"; break;
            case TriggerType.Under    : format = "less than";    break;
        }
        if (!string.IsNullOrEmpty(format))
        {
            format = string.Concat("Cannot be ", format, " '{0}'. \r\n      Current value is
'{1}'.\r\n");
        }
        return (!string.IsNullOrEmpty(format)) ? string.Format(format, this.TriggerValue,
this.PropertyValue) : string.Empty;
    }
}
```

As the class evolved, the constructor became somewhat complex. This occurred mostly due to handling of `DateTime` objects.

There should probably be more code added to verify that the `triggerValue` can be cast to the expectedType (if it is specified), but I'll leave that to the descretion of, and as an exorcise for the programmer (that's you).

```
/// <summary>
/// Constructor
/// </summary>
/// <param name="triggerValue">The value against which a property value wuill be compared for
validity</param>
/// <param name="trigger">The trigger type (default value is TriggerType.Valid)</param>
/// <param name="expectedType">The expected property value type (optional value, default =
null)</param>
public InvalidValueAttribute(object triggerValue, TriggerType trigger=TriggerType.Valid, Type
expectedType=null )
{
    // Note about DateTime properties:
    // Since an attribute constructor parameter must be a constant, we can't specify an actual
    // DateTime object as the trigger value. The code will attempt to accomodate a DateTime by
    // converting the property value to a long, and comparing that to the trigger value.
    // However, you can work around this by actually specifying the expectedType parameter
    // as typeof(DateTime). If you do this, the TriggerValue should also be >= 0, but the
    // constructor will normalize the trigger valu so that it will fall into the acceptable
    // range for a long (Int64).

    if (this.IsIntrinsic(triggerValue.GetType()))
    {
        this.Trigger      = trigger;
        if (expectedType != null)
        {
            if (this.IsDateTime(expectedType))
            {
                // let's try to avoid stupid programmer tricks
                long ticks        = Math.Min(Math.Max(0, Convert.ToInt64(triggerValue)),
Int64.MaxValue);
                // instantiate a datetime with the ticks
                this.TriggerValue = new DateTime(ticks);
            }
            else
            {
                this.TriggerValue = triggerValue;
            }
            this.ExpectedType = expectedType;
        }
        else
        {
            this.TriggerValue = triggerValue;
            this.ExpectedType = triggerValue.GetType();
        }
    }
    else
    {
        throw new ArgumentException("The triggerValue parameter must be a primitive, string, or
DateTime, and must match the type of the attributed property.");
    }

}
```

The last method in the class that I'll be illustrating is the `IsValid` method (since that's what all of this is about).

This method performs the appropriate comparison (as specified by the `Trigger` property) of the property value against the `TriggerValue` specified in the constructor.

```
/// <summary>
/// Determines if the specified value is valid (dependant oin the Trigger property's value).
/// </summary>
/// <param name="value">The value of the property attached to this attribute instance.</param>
/// <returns></returns>
public bool IsValid(object value)
{
    // assume the value is not valid
    bool result = false;
    // save the value for use in the TriggerMsg
    this.PropertyValue = value;
    // get the type represented by the value
    Type valueType = value.GetType();

    if (this.IsDateTime(valueType))
    {
        // ensure that the trigger value is a datetime
        this.TriggerValue = this.MakeNormalizedDateTime();
        // and set the ExpectedType for the following comparison.
        this.ExpectedType = typeof(DateTime);
    }

    // If the type is what we're expecting, we can compare the objects
    if (valueType == this.ExpectedType)
    {
        switch (this.Trigger)
        {
            case TriggerType.Equal    : result = this.IsEqual     (value, this.TriggerValue);
break;
            case TriggerType.Valid    :
            case TriggerType.NotEqual : result = this.IsNotEqual  (value, this.TriggerValue);
break;
            case TriggerType.Over     : result = !this.GreaterThan(value, this.TriggerValue);
break;
            case TriggerType.Under    : result = !this.LessThan   (value, this.TriggerValue);
break;
        }
    }
    else
    {
        throw new InvalidOperationException("The property value and trigger value are not of
compatible types.");
    }
    return result;
}
```

The remainder of the methods in the class are helper methods, type checkers and the actual comparison methods, and really aren't that interesting.

They are only included here in the interest of completeness. Feel free to minimize the following code block to make the article appear shorter.

Because their intended purpose is obvious (to me, at least), I'm not going to explain any of them. Besides, I'm sure by now that you're anxious to tell me why I shouldn't have written this code in the first place.

```
/// <summary>
/// Adjusts the TriggerValue to a DateTime if the TriggerValue is an integer.
/// </summary>
/// <returns></returns>
private DateTime MakeNormalizedDateTime()
{
    DateTime date = new DateTime(0);
    if (this.IsInteger(this.TriggerValue.GetType()))
    {
        long ticks = Math.Min(Math.Max(0, Convert.ToInt64(this.TriggerValue)), Int64.MaxValue);
        date = new DateTime(ticks);
    }
    else if (this.IsDateTime(this.TriggerValue.GetType()))
    {
        date = Convert.ToDateTime(this.TriggerValue);
    }
    return date;
}

#region type detector methods

// These methods can be converted into extension methods, but in keeping with my chosen
// style of code organization, that would have required another file to be created, and
// for purposes of illustration, I didn't feel it was warranted.

// No comments are applied to the methods because I think their functionality is pretty
// obvious by their names.

protected bool IsUnsignedInteger(Type type)
{
    return       ((type != null) &&
                (type == typeof(uint) ||
                type == typeof(ushort) ||
                type == typeof(ulong)));

}

protected bool IsInteger(Type type)
{
    return ((type != null) &&
            (this.IsUnsignedInteger(type) ||
                type == typeof(byte)         ||
                type == typeof(sbyte)        ||
                type == typeof(int)          ||
                type == typeof(short)        ||
                type == typeof(long)));
}

protected bool IsDecimal(Type type)
{
    return (type != null && type == typeof(decimal));
}

protected bool IsString(Type type)
{
    return (type != null && type == typeof(string));
}

protected bool IsDateTime(Type type)
{
    return ((type != null) && (type == typeof(DateTime)));
}

protected bool IsFloatingPoint(Type type)
{
```

```
    return ((type != null) && (type == typeof(double) || type == typeof(float)));
}

protected bool IsIntrinsic(Type type)
{
    return (this.IsInteger(type)       ||
            this.IsDecimal(type)       ||
            this.IsFloatingPoint(type) ||
            this.IsString(type)        ||
            this.IsDateTime(type));
}

protected bool LessThan(object obj1, object obj2)
{
    bool result = false;
    Type objType = obj1.GetType();
    if (this.IsInteger(objType))
    {
        result = (this.IsUnsignedInteger(objType) && this.IsUnsignedInteger(obj2.GetType())) ?
                    (Convert.ToUInt64(obj1) < Convert.ToUInt64(obj2)) :
                    (Convert.ToInt64(obj1) < Convert.ToInt64(obj2));
    }
    else if (this.IsFloatingPoint(objType))
    {
        result = (Convert.ToDouble(obj1) < Convert.ToDouble(obj2));
    }
    else if (this.IsDecimal(objType))
    {
        result = (Convert.ToDecimal(obj1) < Convert.ToDecimal(obj1));
    }
    else if (this.IsDateTime(objType))
    {
        result = (Convert.ToDateTime(obj1) < Convert.ToDateTime(obj2));
    }
    else if (this.IsString(objType))
    {
        result = (Convert.ToString(obj1).CompareTo(Convert.ToString(obj2)) < 0);
    }
    return result;
}

protected bool GreaterThan(object obj1, object obj2)
{
    bool result = false;
    Type objType = obj1.GetType();
    if (this.IsInteger(objType))
    {
        result = (this.IsUnsignedInteger(objType) && this.IsUnsignedInteger(obj2.GetType())) ?
                    (Convert.ToUInt64(obj1) > Convert.ToUInt64(obj2)) :
                    (Convert.ToInt64(obj1) > Convert.ToInt64(obj2));
    }
    else if (this.IsFloatingPoint(objType))
    {
        result = (Convert.ToDouble(obj1) > Convert.ToDouble(obj2));
    }
    else if (this.IsDecimal(objType))
    {
        result = (Convert.ToDecimal(obj1) > Convert.ToDecimal(obj1));
    }
    else if (this.IsDateTime(objType))
    {
        result = (Convert.ToDateTime(obj1) > Convert.ToDateTime(obj2));
    }
    else if (this.IsString(objType))
    {
```

```
        result = (Convert.ToString(obj1).CompareTo(Convert.ToString(obj2)) > 0);
    }
    return result;
}

protected bool IsEqual(object obj1, object obj2)
{
    bool result = false;
    Type objType = obj1.GetType();
    if (this.IsInteger(objType))
    {
        result = (this.IsUnsignedInteger(objType) && this.IsUnsignedInteger(obj2.GetType())) ?
                    (Convert.ToUInt64(obj1) == Convert.ToUInt64(obj2)) :
                    (Convert.ToInt64(obj1) == Convert.ToInt64(obj2));
    }
    else if (this.IsFloatingPoint(objType))
    {
        result = (Convert.ToDouble(obj1) == Convert.ToDouble(obj2));
    }
    else if (this.IsDecimal(objType))
    {
        result = (Convert.ToDecimal(obj1) == Convert.ToDecimal(obj1));
    }
    else if (this.IsDateTime(objType))
    {
        result = (Convert.ToDateTime(obj1) == Convert.ToDateTime(obj2));
    }
    else if (this.IsString(objType))
    {
        result = (Convert.ToString(obj1).CompareTo(Convert.ToString(obj2)) == 0);
    }
    return result;
}

protected bool IsNotEqual(object obj1, object obj2)
{
    return (!this.IsEqual(obj1, obj2));
}

#endregion  type detector methods
```

Example Usage

In order to exercise my attribute, I wrote the following class. The idea is that after the properties are set in my model, I can check the `IsValid` property to make sure I can save the imported object to the database. For purposes of example, I only set four attributed properties. Anything NOT decorated with atttribute is not included in the `IsValid` code.

The first thing you see are the properties that are actually part of the model, and that are decorated with our attribute (refer to the comments for a description of what's going on).

```
// DateTime(JAN 01 2000).Ticks - to fulfill the "constant expression" requirement
// for attribute parameters.
public const long TRIGGER_DATE = 630822816000000000;
// just playing around
public const string TRIGGER_STRING = "ERROR";

/// <summary>
/// Get/set Prop1 (an integer). A value of -1 is invalid
/// </summary>
[InvalidValue(-1, InvalidValueAttribute.TriggerType.Valid)]
public int Prop1 { get; set; }
```

```
/// <summary>
/// Get/set Prop2 (a double). A value that is not between 5d and 10d (inclusive)
/// is invalid. Demonstrates the multiple instances of the attribute that can
/// be attached to a property.
/// </summary>
[InvalidValue(5d, InvalidValueAttribute.TriggerType.Under)]
[InvalidValue(10d, InvalidValueAttribute.TriggerType.Over)]
// To test the distinct attributes requirement, uncomment the next line and
// run the application.
//[InvalidValueAttribute(5d, InvalidValueAttribute.TriggerType.Under)]
public double Prop2 { get; set; }

/// <summary>
/// Get/set Prop3 (a string). A value of "ERROR" is invalid.
/// </summary>
[InvalidValue(TRIGGER_STRING, InvalidValueAttribute.TriggerType.Valid)]
public string Prop3 { get; set; }

// An attribute argument must be a constant expression, typeof expression or array
// creation expression of an attribute parameter type. This means that for a
// DateTime, you MUST determine the trigger value ahead of time and set a constant
// to the appropriate Ticks value (in this case a constant called TRIGGER_DATE).
// The attribute class will adapt to DateTime comparisons when appropriate.
[InvalidValue(TRIGGER_DATE, InvalidValueAttribute.TriggerType.Over, typeof(DateTime))]
public DateTime Prop4 { get; set; }
```

Next, we see the properties that aren't part of the actual model, but that are used to assist the sample application display the results.

```
//////////////////////////
// These properties are not evaluated in the IsValid property code because they're
// not decorated with the InvalidValueAttribute attribute. They are simply used for
// controlling the console output or the validation code.

/// <summary>
/// Get the trigger date value as a DateTime (used to display TRIGGER_DATE constant
/// as a DateTime in the console output.
/// </summary>
public DateTime TriggerDate { get { return new DateTime(TRIGGER_DATE); } }

/// <summary>
/// Get/set the flag that indicates that the IsValid property should short-circuit
/// to false on the first invalid property detected (just a helper for the demo app).
/// If this is false, all of the invalid property error meessages will be appended to
/// the InvalidPropertyMessage string for display in the console.
/// </summary>
public bool ShortCircuitOnInvalid { get; set; }

/// <summary>
/// Get/set the InvalidPropertyMessage.
/// </summary>
public string InvalidPropertyMessage { get; private set; }
```

Finally, we get to the reason we're all here - the `IsValid` property. This property retrieves all properties decorated with the `InvalidValueAttribute` attribute, and processes each attribute instance for each property. Of course, if you don't allow your own custom attribute to have multiple instances, then you can skip the foreach code, but for my purposes, I needed to do this. I also have the requirement that each instance use a unique `Trigger` enumerator, because having multiples of any given `Trigger` doesn't make sense for me.

```
/// <summary>
/// Gets a flag indicating whether or not this object is valid. Validity is determined by
/// properties decorated with InvalidValueAttribute and their associated attribute
/// parameters. Only attributed properties are validated.
/// </summary>
public bool IsValid
{
    get
    {
        // Reset the error message
        this.InvalidPropertyMessage = string.Empty;

        // Assume this object is valid
        bool isValid = true;

        // Get the properties for this object
        PropertyInfo[] infos = this.GetType().GetProperties();
        foreach(PropertyInfo info in infos)
        {
            // Get all of the InvalidValueAttribute attributes for the property.
            // We do this because the attribute is configured to allow multiple
            // instances to be applied to each property. This allows us to
            // specify a more versatile array of error conditions that must all
            // be valid for a given property.
            var attribs = info.GetCustomAttributes(typeof(InvalidValueAttribute), true);

            // if we have more than one attribute
            if (attribs.Count() > 1)
            {
                // make sure they're all distinct (we don't want to allow more than one of
                // each TriggerType)
                var distinct =
attribs.Select(x=>((InvalidValueAttribute)(x)).Trigger).Distinct();
                // If the number of attributes found is not equal to the number of distinct
                // attributes found, throw an exception.
                if (attribs.Count() != distinct.Count())
                {
                    throw new Exception(string.Format("{0} has at least one duplicate
InvalidValueAttribute specified.", info.Name));
                }
            }

            // Now we validate with each attribute in turn.
            foreach(InvalidValueAttribute attrib in attribs)
            {
                // Get the property's value
                object value         = info.GetValue(this, null);

                // See if the property is valid
                bool   propertyValid = attrib.IsValid(value);

                // If it's not valid
                if (!propertyValid)
                {
                    // The object itself isn't valid
                    isValid = false;

                    // Create the error message for this property and add it to the objects
                    // error message string
                    this.InvalidPropertyMessage = string.Format("{0}\r\n{1}",
this.InvalidPropertyMessage,
                                                string.Format("{0} is invalid. {1}",
info.Name, attrib.TriggerMsg));

                    // If we want to short circuit on the first error, break here.
```

```
                    if (this.ShortCircuitOnInvalid)
                    {
                        break;
                    }
                }
            }

        }
        // return the object's isValid status.
        return isValid;
    }
```

Finally, the application is nothing more than a console application that instantiates the sample class, and feeds the console with status messages as the invalid properties are fixed.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FunWithAttributes
{
    class Program
    {
        static void Main(string[] args)
        {
            // create an instance of MyClass with some default values.
            MyModel model = new MyModel()
            {
                Prop1 = 0, // valid (invalid if set to -1)
                Prop2 = 16d,  // valid (invalid if not between 5 and 10 (inclusive))
                Prop3 = "text", // valid (invalid if set to "ERROR")
                Prop4 = DateTime.Now, // invalid if later than 01 January 2000 (see notes regarding DateTimes)
                ShortCircuitOnInvalid = false // so we can see all invalid properties in the console
            };

            try
            {
                //----------------
                Console.WriteLine("Initial configuration:");
                if (model.IsValid)
                {
                    Console.WriteLine("Object is valid");
                }
                else
                {
                    Console.WriteLine(model.InvalidPropertyMessage);
                }
                //----------------
                model.Prop4 = model.TriggerDate.AddDays(-1);
                Console.WriteLine("After correcting Prop4 (a DateTime):");
                if (model.IsValid)
                {
                    Console.WriteLine("Object is valid");
                }
                else
                {
                    Console.WriteLine(model.InvalidPropertyMessage);
                }
                //----------------
                model.Prop2 = 7d;
                Console.WriteLine("After correcting Prop2 (a double):");
                if (model.IsValid)
                {
                    Console.WriteLine("Object is valid");
                }
                else
                {
                    Console.WriteLine(model.InvalidPropertyMessage);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

            Console.ReadLine();
        }
    }
}
```

What I Learned Along the Way

As it is with most programming efforts, I learn little things along the way. At my age, I'll probably forget them by the following day's breakfast,...