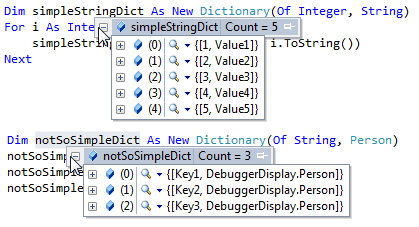
**DebuggerDisplay and DebuggerTypeProxy Attributes**

**Improving visualization of types during debugging sessions**

Tags: Debugging, .NET, Visual Studio, C#, VB.NET

***The code is available for download INSERT DOWNLOAD LINKS HERE***

There are a “few” projects I’ve wanted to do for a long time but this subject has become something I only recently started implementing fairly often. The need for readier access to the contents of my object during execution time arose while working with a custom type containing several Dictionaries. The out-of-the-box VS implementation starts to falter as soon as you stray from the primitives like “Dictionary(Of Integer, String)”. Luckily I remembered a Tech Days presentation where the DebuggerDisplay attribute was one of the things discussed. The display of the Integer to String dictionary is pretty good, but once you go to a String to Person dictionary, or any other custom type really, things are much less impressive. They seem to be all DebuggerDisplay.Persons, really…



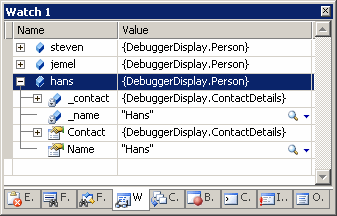
We will be using a simple Person class (containing properties Name and Age) and start improving the debugging experience with DebuggerDisplay and other attributes till we get to DebuggerTypeProxy which allows you to hijack the entire visualization of your types during debug. With Visual Studio 2010 the debug display of KeyValuePairs has improved but as we’ve already seen, when working with custom types you are still pretty much in the dark till you start digging for the useful bits of Person information.

DebuggerDisplayAttribute

The DebuggerDisplayAttribute, to be found in the System.Diagnostics namespace, allows you to override the representation of instances of your own types during debugging in Visual Studio.

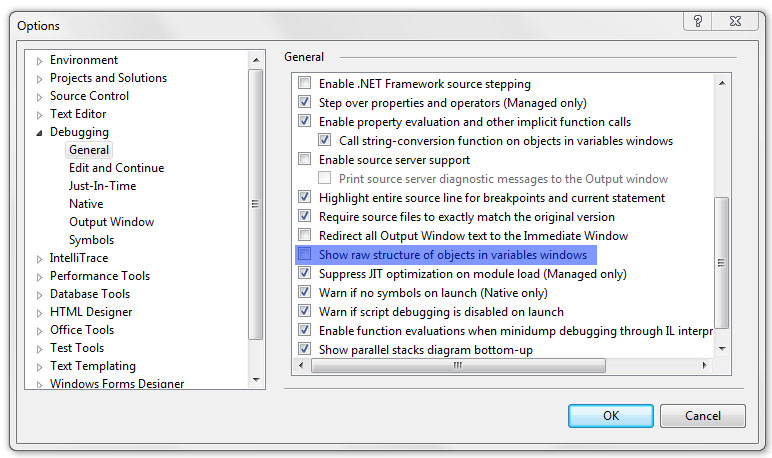
The default debug presentation is the Object.ToString() implementation:

Return Me.GetType().ToString()



Which prints the Type.FullName and is not very helpful. In C# you can override ToString and your implementation will be used in the debug windows. DebuggerDisplay is there for you when you have already overridden ToString and want to see different data during debugging sessions and for when you are using a different language than C#.

**Quick Tip**: If your DebuggerDisplay implementations don’t show up, you might want to look at the “Show raw structure of objects in variables windows” setting under Tools, Options, Debugging, General.



Obviously, I’m not the first to blog about the DebuggerDisplay attribute and I especially enjoyed the entry on [jaredpar's Weblog](http://blogs.msdn.com/b/jaredpar/archive/2011/03/18/debuggerdisplay-attribute-best-practices.aspx) (<http://blogs.msdn.com/b/jaredpar/archive/2011/03/18/debuggerdisplay-attribute-best-practices.aspx>) where he wrote about possible performance issues during debugging when using the attribute too creatively and where he suggested the following pattern:

[DebuggerDisplay("{DebuggerDisplay,nq}")]

public class Student

{

public string FirstName { get; set; }

public string LastName { get; set; }

private string DebuggerDisplay

{

get { return string.Format("Student: {0} {1}", FirstName, LastName); }

}

}

I like this approach very much, it allows for a lot of flexibility, such as debugging the DebuggerDisplay while running your application! I’m now using this “pattern” constantly for [DTO](http://en.wikipedia.org/wiki/Data_Transfer_Object)’s (<http://en.wikipedia.org/wiki/Data_Transfer_Object>) and more.

**Quick Tip:** The **,nq** part in the DebuggerDisplay attribute will remove the quotes around the resulting string (,nq = No Quotes).

Keeping the property private keeps it from polluting the public interface but I’m not sure how to best handle the pattern when inheritance is involved. Making the property protected virtual might confuse someone inheriting from your class who is unfamiliar with the pattern and/or attribute. Assigning the same DebuggerDisplay property code to both base and derived classes opposes the **Pragmatic Tip 11: Don’t Repeat Yourself** ([DRY](http://en.wikipedia.org/wiki/Don't_repeat_yourself)). If all you want to do is add a little extra to the original implementation. I tend to refine and improve my original display while debugging and going through several files making the same change is something I just don’t want to waste time on. So I’ve kept the property private but have overridden ToString to return DebuggerDisplay and call that from the derived implementations. I’m not very happy with this solution however. (And I can’t merely override ToString as it is a VB.NET project)

public class SuperHero : Person

{

public string Power { get; set; }

public SuperHero(string name, int age, string power)

: base(name, age)

{

Power = power;

}

[DebuggerBrowsable(DebuggerBrowsableState.Never)]

protected override string DebuggerDisplay

{

get

{

return string.Format("{0} ({1})", base.DebuggerDisplay, Power);

}

}

}

Most online examples I’ve seen decorated a class and the attribute is definitely most useful when decorating classes and structs but according to the [MSDN documentation](http://msdn.microsoft.com/en-us/library/system.diagnostics.debuggerdisplayattribute.aspx) (<http://msdn.microsoft.com/en-us/library/system.diagnostics.debuggerdisplayattribute.aspx>) it can also be placed on Fields, Enums, Properties, Delegates and Assemblies. Let’s get busy with the last one.

Assembly

As [The Moth](http://www.danielmoth.com/Blog/Use-The-DebuggerDisplayAttribute-For-Your-Types.aspx) (<http://www.danielmoth.com/Blog/Use-The-DebuggerDisplayAttribute-For-Your-Types.aspx>) blogged, you can add your own DebuggerDisplay attribute to classes of which you do not own the source code by creating a Class Library with the attribute assigned in the AssemblyInfo and placing it in the VS.NET visualizers directory (%userprofile%\Documents\Visual Studio 20XX\Visualizers).

C#  
[assembly: System.Diagnostics.DebuggerDisplay(

"MemoryUsage=\\{{Length,nq} / {Capacity,nq}\\} ({(Length/(double)Capacity).ToString(\"p\"),nq})",

Target = typeof(System.Text.StringBuilder))]

VB.NET in VS2008

<Assembly: System.Diagnostics.DebuggerDisplay("MemoryUsage={{{Length,nq} / {Capacity,nq}}}",

Target:=GetType(System.Text.StringBuilder))>

VB.NET in VS2010 (breaking change!?)  
<Assembly: System.Diagnostics.DebuggerDisplay("MemoryUsage=\{{Length,nq} / {Capacity,nq}\} ({Math.Round(Length / Capacity \* 10000) / 100,nq}%)",

Target:=GetType(System.Text.StringBuilder))>

I couldn’t quite get ToString(“p”) to work in VB.NET. DebuggerDisplay("MemoryUsage=\{{Length,nq} / {Capacity,nq}\} ({(Length / Capacity).ToString(""p""),nq})") looks like it should work, but it doesn’t. Hence the workaround ☺

**Quick Tip**: You have to restart VS.NET for the visualizer to become active!  
  
If you add the attribute to a type that already has one defined, you will overrule the original implementation while keeping the benefits of the DebuggerTypeProxy, if one is available. Later in this article we will revisit the debugdisplay of StringBuilder by using such a proxy but let’s move on to applying DebuggerDisplay to an enum first.

Enumerations

When applying the attribute to an Enumeration we could, for example, display both the name and the value during debugging. Casting the {this} to an int to obtain the numerical value is something that only seems to work in C#, I tried all sorts of variations in VB.NET and ended up with: DebuggerDisplay("{ToString(),nq}={CInt([Enum].Parse(GetType(ContactTypes), ToString()))}"), but didn’t get it to work, probably due the Enum.Parse part (perhaps for the better). And here is the code, which is what you get as default in VB.NET anyway:

[DebuggerDisplay(@"{this} \{{(int)this,nq}\}")]

public enum ContactTypes

{

Unknown = 0,

Home = 1,

Work = 2,

Billing = 4

}

C:\Users\Wouter\Desktop\DebuggerDisplay\enums\0 simpleContact.png

We could make our ContactTypes give even more detailed information about their status by using Jon Skeets [Unconstrained Melody](http://code.google.com/p/unconstrained-melody/) (<http://code.google.com/p/unconstrained-melody/>) and the System.ComponentModel.DescriptionAttribute to give each element a different description which would show up during debugging sessions. This is a non-generic copy of Jon’s Reflection code in UnconstrainedMelody.EnumInternals<T>.GetDescription(T value). A static Helpers class was created to execute the DebuggerDisplay code since an enumeration cannot contain methods or properties. Do not use this code outside the Melody as executing this with a combined Flags value will throw a NullReferenceException!

public static class Helpers

{

public static string EnumDebuggerDisplay(object value)

{

FieldInfo field = typeof(ContactTypes).GetField(value.ToString());

return field.GetCustomAttributes(typeof(DescriptionAttribute), false)

.Cast<DescriptionAttribute>()

.Select(x => x.Description)

.FirstOrDefault();

}

}

[DebuggerDisplay("{WeblogSpace.Helpers.EnumDebuggerDisplay(this)}")]

public enum ContactTypes

{

[Description("Uknown contacttype")]

Unknown = 0,

[Description("Home address")]

Home = 1,

[Description("Workplace address")]

Work = 2,

[Description("Parental address")]

Billing = 4

}

While Unconstrained Melody addresses an unfortunate ‘constraint’ (or should I say lack thereof) of Generics, I’m not sure if using Description and DebuggerDisplay attributes on enums will improve your debugging sessions. So enough of those enums! What is left are Properties, Delegates and Fields. Before you go about decorating pretty much everything in your types with DebuggerDisplayAttributes, it is good to know that you can keep your code clean from DebuggerDisplay attributes while still achieving the debug experience you’ve always dreamed of. It’s time to introduce DebuggerTypeProxy.

***Do a small test with fields, properties and delegates! Probably nothing interesting is to be said but check it out anyway***

***Create a code snippet to generate the DebuggerDisplay code***

DebuggerTypeProxy

To come back to our assembly DebuggerDisplay example to modify the visualization of the StringBuilder,

***(eerste x lijnen tonen met rootexpanded)***

***ILSpy: check alle attributes in System.ComponentModel (en die van Forms)  
koop refactor aan?***

***Sla alles op op externe HD en op dropbox***

***TEST STUFF:***

* Conditional(“DEBUG”) check it with return value?
* RootHidden & Never attributes
* Try DebuggerDisplay on property
* Generic Dictionary proxy
* ExpandoObject
* References (naar gebruikte blogs etc)
* ExpandableTypeConverter

Public Class Person

Private \_name As String

Private \_contact As ContactDetails

Public ReadOnly Property Name() As String

Get

Return \_name

End Get

End Property

Public ReadOnly Property Contact() As ContactDetails

Get

Return \_contact

End Get

End Property

Public Sub New(ByVal name As String, ByVal contact As ContactDetails)

\_name = name

\_contact = contact

End Sub

End Class

Public Class ContactDetails

Private \_email As String

Private \_telephone As String

Public ReadOnly Property Email() As String

Get

Return \_email

End Get

End Property

Public Sub New(ByVal email As String, ByVal tel As String)

\_email = email

\_telephone = tel

End Sub

End Class

Conditional(“???”)

, both available in the System.Diagnostics namespace,

References:

Things to learn:

* T4 Templates
* NHibernate <-> Entity Framework
* UnitTesting
* VS.NET Macro’s
* Programming VS.NET