Dismembering MemberData

Doug Hennig

A couple of common requests from VFP developers are capitalization support for custom properties and methods in the Properties window and IntelliSense and the ability to customize the Properties window. Microsoft listened: VFP 9 gives us both of these, plus the ability to create our own property editors, with a new feature called MemberData.

One of the things that has bugged me about VFP since its initial release is that custom properties and methods are forced to lower case in a VCX or SCX file. That means they appear at the bottom of the Properties window rather than interspersed with native properties, events, and methods (referred to as PEMs or members). It also means that IntelliSense doesn't display member names in a meaningful case, so I always end up correcting the name that IntelliSense inserts for me. I've also wished that there was a way to limit the Properties window to just displaying those PEMs that mattered to me, not the hundreds of obscure ones I never use.

VFP 9 has answered my prayers. This new version provides a way to specify meta data about class members. This meta data, referred to as MemberData, contains attributes such as the case used to display a member (the name is still physically stored in the VCX or SCX in lower case) and whether to display a member in the new Favorites tab in the Properties window.

MemberData is implemented by adding a new property to a class called _MemberData, and filling it with XML that contains the meta data for the members of the class. _MemberData isn't added to a class automatically, nor is the XML filled in; you have do both of those yourself (although later in this article we'll address this).

MemberData

Here's the XSD schema for MemberData:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="VFPData">
<xs:complexType>
<xs:sequence>
<xs:element name="memberdata">
<xs:complexType>
<xs:attribute name="name" type="xs:string"
use="required" />
<xs:attribute name="type" type="xs:string"/>
<xs:attribute name="display" type="xs:string"/>
<xs:attribute name="favorites" type="xs:boolean"/>
<xs:attribute name="override" type="xs:boolean"/>
<xs:attribute name="script" type="xs:string"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Okay, now here it is in English <g>. Table 1 shows the attributes that make up a member's meta data.

Table 1. The attributes in MemberData contain the meta data for a member.

Attribute	Description
name	The name of the member
type	The member type: "property", "event", or "method"
display	The text to use as the name of the member in the Properties window and in IntelliSense
favorites	"True" if the member should appear in the Favorites tab of the Properties window or "False" if not
override	"True" to ignore meta data from a parent class or "False" to inherit from a parent class

script	Code to execute when the property editor button for this member is selected in the Properties window

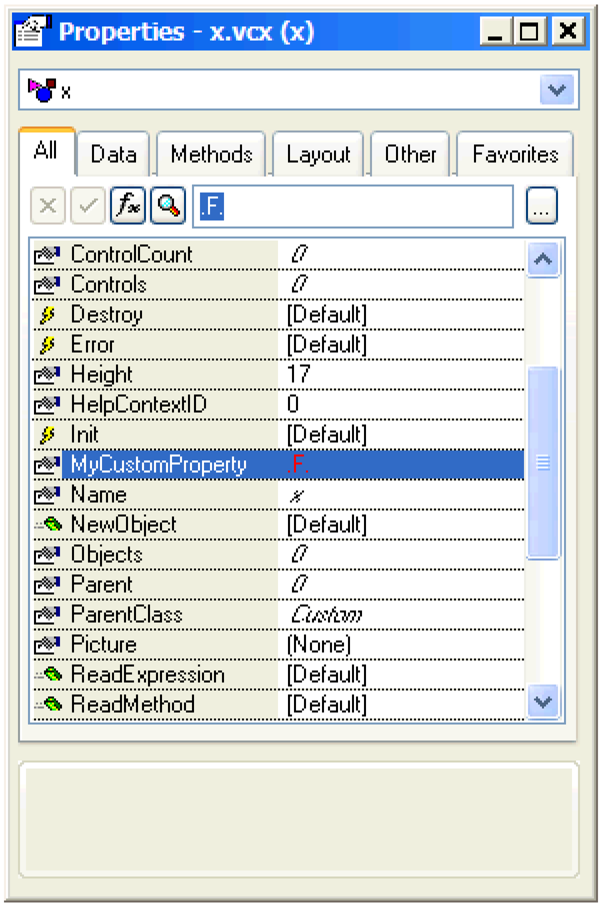
Here are some notes about these elements:

- Since this is XML, the names of the elements and attributes are case-sensitive. With the exception of the script attribute, the contents of the attributes are also case-sensitive. The name and type attributes must be in lower case. “True” and “False” must be specified in proper case in the favorites and override attributes.
- The display attribute cannot be different from the name of the member except in case. For example, while “MyCustomProperty” is acceptable for display for the mycustomproperty member, “SomeOtherName” is not.
- You can modify MemberData in subclasses without having to reproduce the entire XML string; unspecified attributes get their values from the parent class. However, if the override attribute is “True”, the default behavior is used for unspecified attributes instead. For example, suppose you have a class that has meta data specifying a display attribute for MyCustomProperty. If the display attribute is omitted in a subclass, it will inherit the display attribute from the parent class. However, if override is “True”, the property will display as “mycustomproperty” in the Properties window because the display attribute isn’t inherited from the parent class and isn’t specified in this class.
- If the script attribute is specified for a member, the Properties window includes a property editor button when the member is selected. Clicking on that button executes the code specified in the script attribute using EXECSCRIPT().
- If the XML isn’t valid, you won’t get an error, but you won’t see the effects of MemberData either.

Here’s an example specifying that the property mycustomproperty is displayed as MyCustomProperty, it appears in the Favorites tab of the Properties window, and VFP calls MyCustomPropertyEditor.PRG when you click on the property editor button in the Properties window.

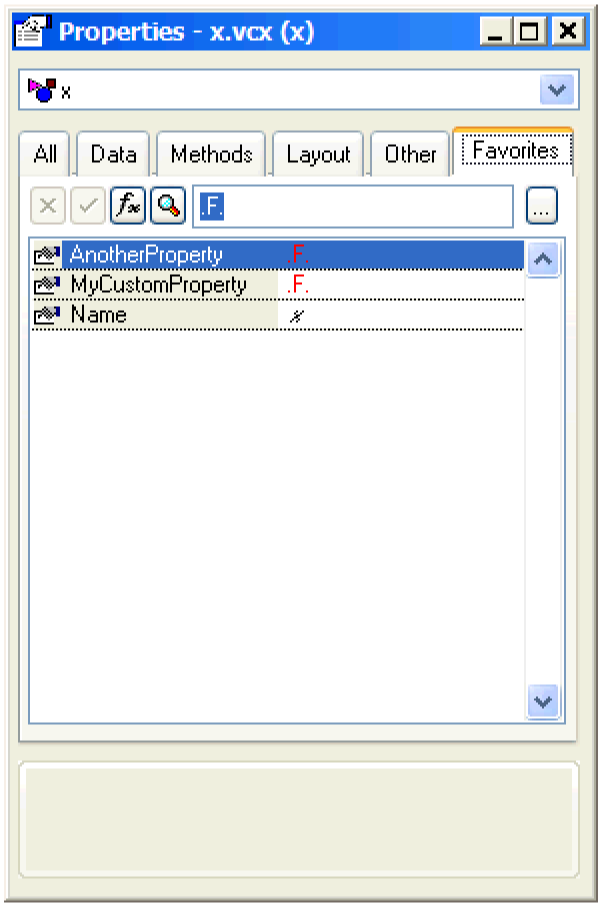
```
<?xml version = "1.0" encoding="Windows-1252"
standalone="yes"?>
<VFPData>
<memberdata
name="mycustomproperty"
type="property"
display="MyCustomProperty"
favorites="True"
override="False"
script="DO MyCustomPropertyEditor.PRG"/>
</VFPData>
```

Figure 1 shows how this property appears in the Properties window. Notice that it appears as “MyCustomProperty”, is in the proper place in alphabetical order rather than at the end of the Properties window, and that a property editor button appears to the right of the value textbox. In Figure 2, you can see that this property is one of the few shown in the Favorites tab. Figure 3 shows how the property appears in IntelliSense.



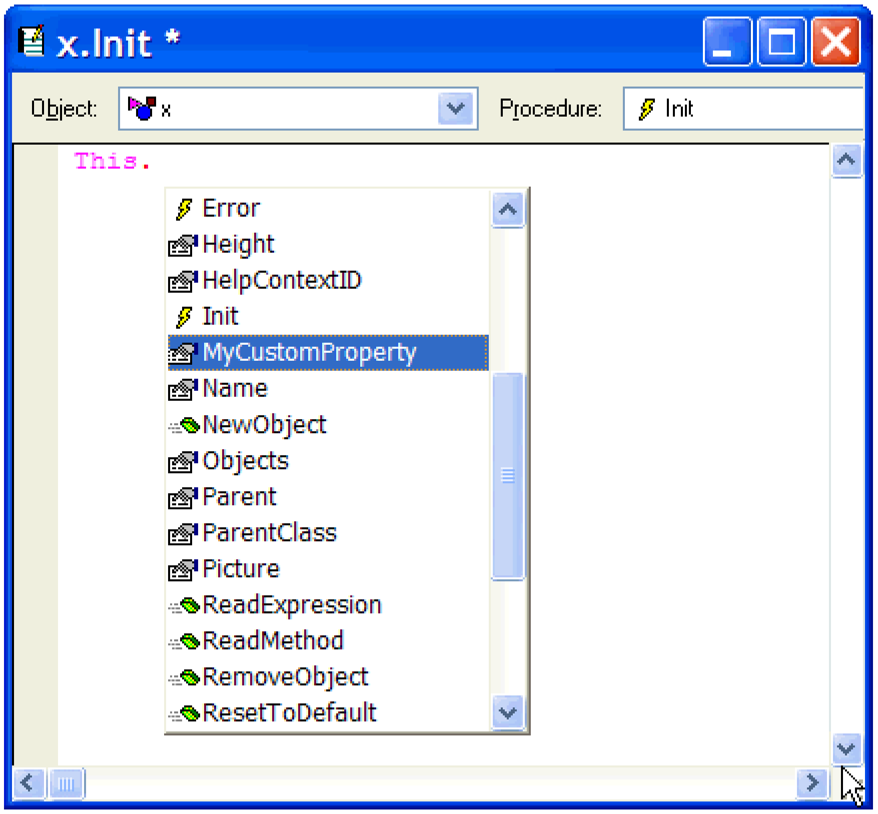
ControlCount	0
Controls	0
Destroy	[Default]
Error	[Default]
Height	17
HelpContextID	0
Init	[Default]
MyCustomProperty	.F.
Name	x
NewObject	[Default]
Objects	0
Parent	0
ParentClass	Custom
Picture	(None)
ReadExpression	[Default]
ReadMethod	[Default]

Figure 1. Thanks to MemberData, MyCustomProperty appears the way we want it to in the Properties window.



AnotherProperty	.F.
MyCustomProperty	.F.
Name	x

Figure 2. The Favorites tab allows you to see only those properties you consider important.



```
This.
```

Figure 3. IntelliSense respects MemberData so members appear in the correct case.

Global MemberData

Having class-specific MemberData is great, but it would be a drag if you had to specify the same meta data for the same property in every class. For example, if you add MyCustomProperty to several classes, you'd likely want to use the same MemberData for each class. Fortunately, the VFP team thought of this. To create MemberData for a member at a global level (that is, for all classes having that member), add a record to the IntelliSense table that specifies the MemberData. (The IntelliSense table, specified in the _FOXCODE system variable, defaults to FOXCODE.DBF in your application data directory, which is the one returned by the HOME(7) function.) Set the TYPE field to "E", a new value in VFP 9 denoting a MemberData record, put the name of the member into ABBREV (case isn't important), and put the MemberData into TIP. If the _MemberData property for an object contains meta data for a member that also has global meta data, the _MemberData version will override the global.

Here's an example that creates global MemberData for MyCustomProperty by adding a record with the proper information to FOXCODE:

```
local lcXML

* Create the MemberData.

text to lcXML noshow
<?xml version="1.0" encoding="Windows-1252"
standalone="yes"?>
<VFPData>
<memberdata
name="mycustomproperty"
type="property"
display="MyCustomProperty"
favorites="True"
override="False"
script="do MyCustomPropertyEditor.PRG"/>
</VFPData>
endtext
```

```
* Now create a record in FOXCODE that provides the
* MemberData for MyCustomProperty.

use (_foxcode) again shared alias FOXCODE
insert into FOXCODE (TYPE, ABBREV, TIP) values ('E', ;
  'MyCustomProperty', lcXML)
use
```

Check this out: run this code (GlobalMemberData.PRG in this month's Subscriber Downloads), then type the following in the Command window:

```
create class x of x as custom
```

Create a property called MyCustomProperty, and notice that even though there's no _MemberData property for this class, MyCustomProperty appears in the proper case and in the Favorites tab.

MemberData doesn't just apply to custom properties and methods. For example, since Caption and Name are the properties I change most often for Form, Label, Checkbox, and other objects with those properties, I've created global MemberData for those properties to add them to the Favorites tab so I don't have to jump around in the Properties window to set them. This gives me a small productivity boost for every object, and over the life of a project, it can really add up.

Property editors

Like a builder, a property editor can make it easier to enter the value of a property. For example, the new Anchor property defines how a control reacts when its container is resized (for example, it could be resized or moved). However, the Anchor value consists of additive enumerations, such as 12 (8 = anchored to the right + 4 = anchored to the bottom). This isn't exactly intuitive, so a property editor would help with productivity.

Although you can specify multiple lines of code in the meta data script attribute, calling a PRG or form usually makes more sense. For global MemberData, you can also specify that you want to execute a script record in FOXCODE by specifying the following as the script attribute:

```
do (_CODESENSE) with 'RunPropertyEditor', '', 'SomeValue'
```

SomeValue is a value you want passed to the script. Put "{ScriptName}" into CMD, where "ScriptName" is the name of the script. To create a script record, add a record to FOXCODE with "S" in TYPE, the name of the script in ABBREV, "{}" in CMD, and the code to execute in DATA. Using a script record has the benefit that it's more compact than a separate property editor (since the code is contained in FOXCODE) and there aren't any pathing issues.

Like a builder, a property editor is responsible for obtaining a reference to the object being edited and for writing to the object's property. In fact, you can think of a property editor as a subset of a builder (which is usually an editor for multiple properties of an object), with similar requirements and architecture issues.

Getting there from here

Now that you know how MemberData works, all you have to do to use it is add an _MemberData property to every object and then fill in the appropriate XML in that property. Doesn't that seem like it'll *reduce* your productivity rather than improve it? Fortunately, there are two things we can do about that: tell VFP to automatically create an _MemberData property for every class and figure out a way to automatically generate the XML we need.

The key to the first task is adding a record to the IntelliSense table that's executed whenever you open a class in the Class Designer or a form in the Form Designer. It turns out that when you open the Class or Form Designers, VFP looks in the IntelliSense table for a record with TYPE set to "E" and ABBREV set to "_GetMemberData". If it finds such a record, it executes the code in the DATA memo field. So, we can add a record to FOXCODE that creates an _MemberData property in any class or form that doesn't already have it. Here's some code that will do this:

```
local lcCode
```

```
* Create the code we want inserted into FOXCODE.

text to lcCode noshow
lparameters toFoxcode
local laObjects[1], ;
  loObject, ;
  lcXML
aselobj(laObjects)
if aselobj(laObjects) = 0 and aselobj(laObjects, 1) = 0
  return ''
endif aselobj(laObjects) = 0 ...
loObject = laObjects[1]
if vartype(loObject) = 'O' and ;
  not pemstatus(loObject, '_memberdata', 5)
  loObject.AddProperty('_memberdata', '')
endif vartype(loObject) <> = 'O' ...
return ''
endtext

* Now create a record in FOXCODE that executes whenever a
* class is opened in the Class Designer.

use (_foxcode) again shared alias FOXCODE
insert into FOXCODE (TYPE, ABBREV, DATA) values ('E', ;
  '_GetMemberData', lcCode)
use
```

The code inserted into the DATA memo uses ASELOBJ() to get a reference to the class or form being edited, checks for the existence of _MemberData with PEMSTATUS(), and adds the property using AddObject if it wasn't found.

To see how it works, run this code (UpdateFoxCode.PRG in this month's Subscriber Downloads), then type the following in the Command window:

```
create class x of x as custom
```

In the Properties window, you'll see that VFP automatically created an _MemberData property for this class.

Now that we have an _MemberData property for anything we open in the Class or Form Designers, how to we tackle the second task, generating the XML for the meta data?

The _MemberData property editor

As I was typing XML for the various properties in a custom class, it struck me that since MemberData can define a property editor for a property, why couldn't we have a property editor for _MemberData itself? We don't have to specify such an editor for every class, since a global record for _MemberData can handle it. This along with the ability to automatically add _MemberData to every class means that now creating meta data for members is a piece of cake.

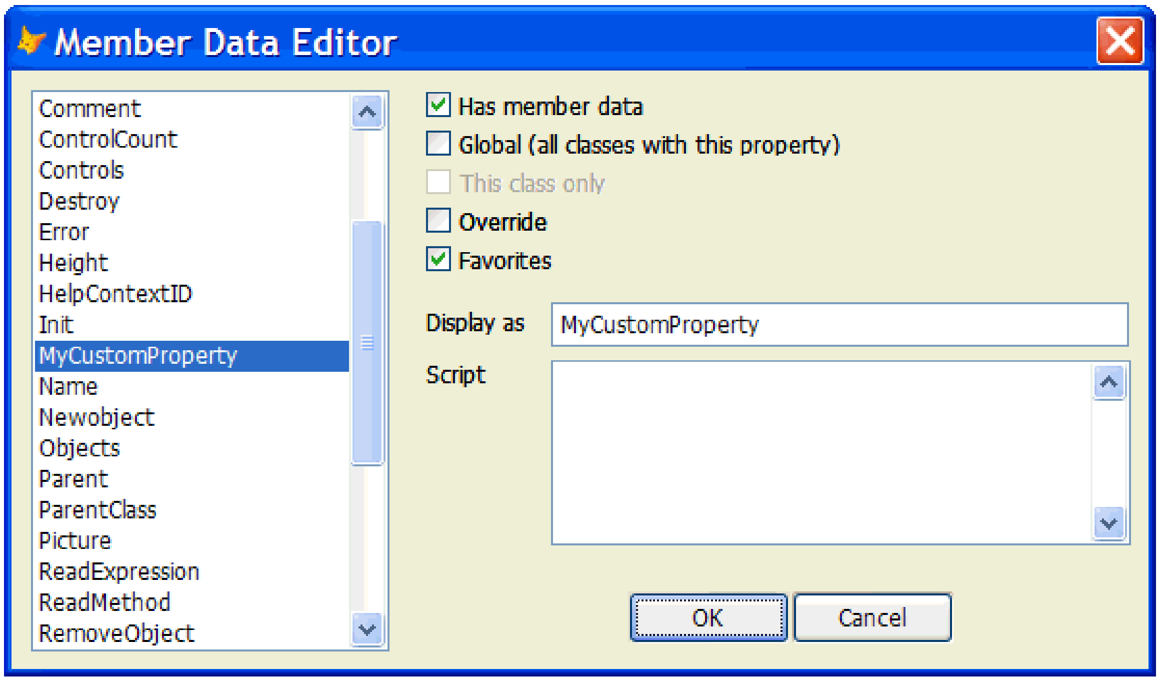
I won't show the code for the MemberData property editor for space reasons, but will explain how it works:

- MemberDataEditor.PRG, which is specified in the script attribute of the global meta data for _MemberData, starts by using ASELOBJ() to get a reference to the object being edited. If there isn't one (for example, you executed it using DO from the Command window), this must be a "setup" call, so it creates the _GetMemberData record described earlier and the global meta data record for _MemberData in FOXCODE and then exits.

- It reads the existing _MemberData property from the object and any global MemberData from FOXCODE.

- It uses AMEMBERS() to create a array of PEMs for the object. These PEMs are then put into a collection of MemberDataObject objects. (MemberDataObject is a class defined in MemberDataEditor.PRG that just consists of properties representing the individual MemberData attributes.) Existing _MemberData and global meta data is included in the information in the collection.

- MemberDataEditor.PRG hides the Properties window (since it may not refresh properly if it's open when the MemberData is edited), and then runs MemberDataEditor.SCX, passing it a reference to the object being edited and the collection of PEMs. This form is shown in Figure 4.
- You use MemberDataEditor.SCX to edit the MemberData for members. Select a member from the list, turn on the Has Member Data setting, and set the desired attributes. If you turn on the Global setting, the meta data for the selected member will be written to a record in FOXCODE rather than _MemberData so the meta data is global. If the selected member has global meta data, use the This Class Only setting to indicate that any attributes you change apply to this class only (in other words, the meta data will be written to _MemberData rather than FOXCODE). When you're finished changing the members you want meta data for, click on the OK button.
- The Click method of OK calls the form's SaveMemberData method. This method creates the XML string for the meta data. It writes the XML for any members that have Global turned on and This Class Only turned off to FOXCODE, and combines the XML for all other members, writing it to the _MemberData property of the object being edited.
- Finally, MemberDataEditor.PRG redisplays the Properties window if it was visible before.


Member Data Editor

Comment
ControlCount
Controls
Destroy
Error
Height
HelpContextID
Init
MyCustomProperty
Name
Newobject
Objects
Parent
ParentClass
Picture
ReadExpression
ReadMethod
RemoveObject

[✓] Has member data
[ ] Global (all classes with this property)
[ ] This class only
[ ] Override
[✓] Favorites

Display as: MyCustomProperty

Script:

Figure 4. The MemberData property editor makes short work of generating the XML for member meta data.

To try this out, DO MemberDataEditor.PRG to set it up properly, then create or modify a class in the Class Designer. Add some custom properties and methods, then select _MemberData in the Properties window and click on the property editor button. Modify the MemberData for the custom properties and methods so they display with the proper capitalization, and turn on Favorites for some of them. Click on the OK button and notice that these members now appear the way you wish in the Properties window.

Summary

MemberData is a great productivity enhancement in VFP 9 because it allows you to specify how members should be displayed in the Properties window and IntelliSense, making them easier to find and saving editing if you let IntelliSense insert their names into code. Also, like builders, property editors can make it much easier to put the proper values into a property. I expect we'll see a number of property editor ship with VFP 9, and many others will be created by enterprising VFP developers.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com