Favorites for IntelliSense

Doug Hennig

Favorites for IntelliSense is a new tool that extends IntelliSense, allowing you to specify which members of a class you want displayed in IntelliSense, as well as easily scripting tooltips or lists of values for properties or parameters of methods.

IntelliSense is easily the best feature ever added to Visual FoxPro. It provides a greater productivity boost to VFP developers than anything added before or since. However, one thing that bugs me about IntelliSense is that when used with a class, it displays all members of that class rather than the ones I really want to see.

For example, **Figure 1** shows the IntelliSense display for the ConnectionMgr class. Note that although there are only a few custom properties and methods we're interested in, IntelliSense displays everything. This requires more effort to select the exact member you want, especially if you're not very familiar with the class.

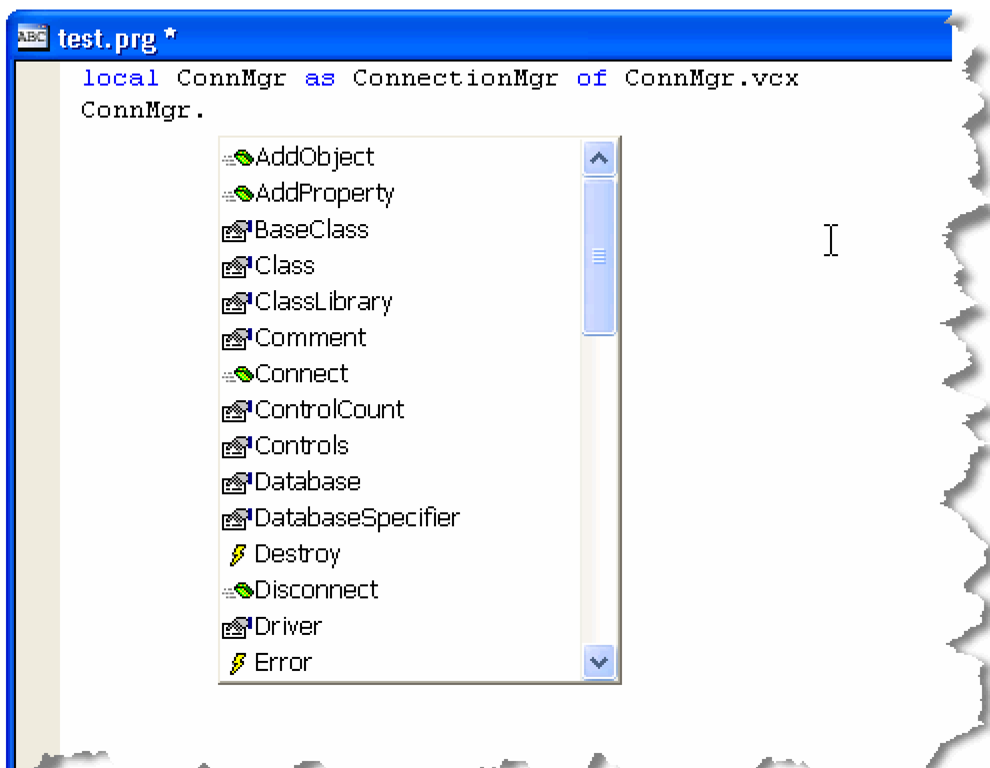

Figure 1. Although IntelliSense allows you to choose a member name from a list, it displays more items than we usually need.

We had a similar problem with the Properties window until VFP 9. In that version, Microsoft added a Favorites tab that displays only those members you defined as belonging on that tab. (One way to do that is to right-click the member in the Properties window and choose Add to Favorites). So now, rather than wading through dozens of members to find the one you want, you can pick among the few you've specified on the Favorites tab.

What I'd really like to see is the equivalent of the Favorites tab for IntelliSense. So, I decided to create it. The result is Favorites for IntelliSense (FFI).

Favorites for IntelliSense

Before we look under the hood, let's see FFI in action. The first thing we need to do is register a class with FFI. Open the ConnectionMgr class in ConnMgr.VCX (found in this month's Subscriber Downloads), then type DO FORM FFIBUILDER in the Command window. You should see the dialog shown in **Figure 2**. This dialog allows you to specify the "namespace" for the class. The namespace is the name that appears in the IntelliSense list when you type LOCAL SomeVariable AS. It defaults to the name of the class, but you can specify something else if you wish. For example, for a class named cApplication, you may want the namespace to be Application instead. The description is used as the tooltip for the class in the IntelliSense list if this class is used as a member of another class (for example, the User property of the sample cApplication class contains an instance of the cUser class, so that class' description is used for the User property). It defaults to the description for the class as specified in the Class Info function in the Class menu or by choosing the Edit Description function in the Project menu when the class is selected in the Project Manager.

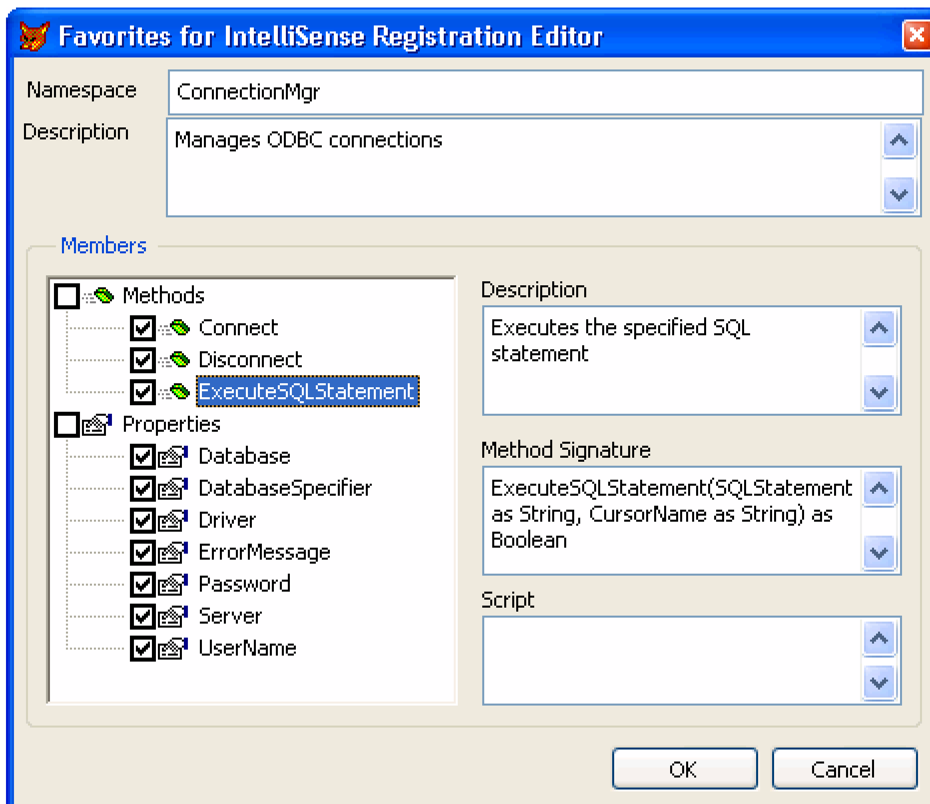

Figure 2. The Favorites for IntelliSense builder allows you to specify what appears in IntelliSense for the selected class.

The TreeView shows public custom properties and methods for the class; if you want native members displayed as well, change the AMEMBERS() statement in the LoadTree method of the FFIBuilderForm class in FFI.VCX. The checkbox before the name indicates whether the member is included in IntelliSense or not; by default, all custom members are included. The description is used as the tooltip for the member in the IntelliSense list; it defaults to the description you entered for the member when you created it. The method signature is displayed as a tooltip for a method when you type an open parenthesis or a comma in the parameter list for the method; this tooltip shows you what parameters can be passed to the method. The signature defaults to the method name and the contents of any LPARAMETERS statement in the method, but you can edit it to display anything you wish, including the data type of the return value. The script editbox allows you to enter code that's executed when you select this member and type an open parenthesis, a comma in the parameter list, or an equals sign (to assign a value to a property). We'll see an example of this later.

Make any changes you wish in this dialog, then click OK. This form adds records for the class and each of the registered members to an FFI table. It also adds two records to your IntelliSense table: one for the class and one for an IntelliSense script for the class. We'll look at this in more detail later as well.

To see how Favorites for IntelliSense works with this class, create a PRG and type LOCAL ConnMgr AS. When you press Space after AS, you should see the IntelliSense list of types shown in **Figure 3**. When you select ConnectionMgr and press Enter, you'll see the following code inserted into the PRG (where Path is the path for the VCX):

```
local ConnMgr as ConnectionMgr
ConnMgr = newobject('Connectionmgr', ;
  'Path\connmgr.vcx')
```

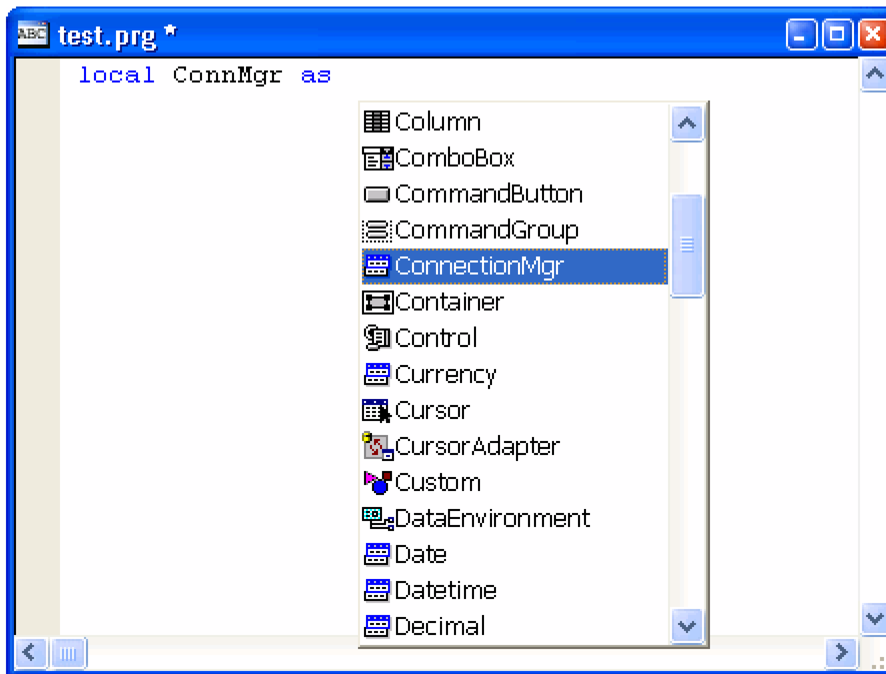

Figure 3. Any class registered with Favorites for IntelliSense is displayed as a type in IntelliSense.

Now type ConnMgr followed by a period. As you can see in **Figure 4**, IntelliSense only displays registered members of the class; hence "Favorites for IntelliSense." If you select a method such as ExecuteSQLStatement, when you type the opening parenthesis, you'll see the method signature as the IntelliSense tooltip, making it much easier to see what parameters to pass to the method.

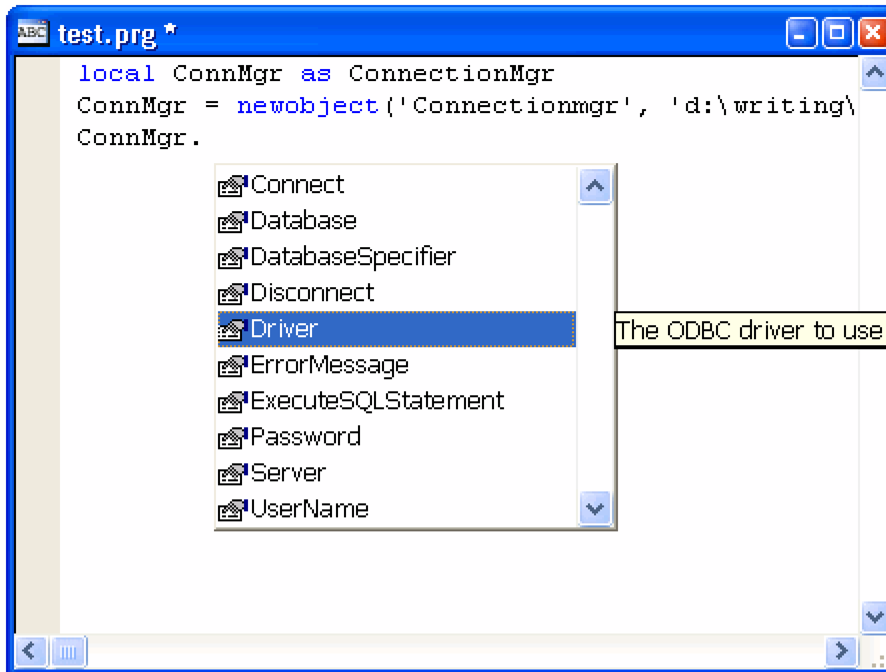

Figure 4. IntelliSense only displays the registered members for the specified class; compare this to Figure 1.

How it works

The secret behind FFI lies in two things: how IntelliSense deals with things defined as "types" in the IntelliSense table and IntelliSense scripts. Types are normally use for data types, such as Integer or Character, and base classes, such as Checkbox and Form. However, other things can be defined as types as well, either by manually adding records with TYPE set to "T" or using the IntelliSense Manager in the Tools menu. Other kinds of type records are usually custom classes or COM objects so you get IntelliSense on them. That's what we'll use type records for as well, but we'll customize how IntelliSense works using a script and a custom IntelliSense-handling class.

If you look in your IntelliSense table (USE (_FOXCODE) AGAIN and BROWSE) after using the FFIBUILDER form to register a class, you'll see two new records at the end of the table. One is the type record for the class; it doesn't contain much information other than the namespace you defined in the builder form and the name of a script to use for IntelliSense for the class (that's specified as a name surrounded by curly braces in the CMD column). The other is a script record, with TYPE set to "S" and ABBREV containing the name specified in the CMD column of the T record. The name is FFI_ followed by a SYS(2015) value; because the field is short, there wasn't enough room to use something more descriptive such as "Namespace_Script."

The script record has the following code in its DATA memo (in this code, Path1 is replaced with the path for FFI.VCX and Path2 is replaced with the path for the VCX of the registered class):

```
lparameters toFoxCode
local loFoxCodeLoader, ;
  luReturn
if file(_codesense)
  set procedure to (_codesense) additive
  loFoxCodeLoader = createobject('FoxCodeLoader')
  luReturn        = loFoxCodeLoader.Start(toFoxCode)
  loFoxCodeLoader = .NULL.
  if atc(_codesense, set('PROCEDURE')) > 0
    release procedure (_codesense)
  endif atc(_codesense, set('PROCEDURE')) > 0
else
  luReturn = ''
endif file(_codesense)
return luReturn
```

```
define class FoxCodeLoader as FoxCodeScript
  cProxyClass    = 'FFIFoxCode'
  cProxyClasslib = 'Path1\ffi.vcx'

  procedure Main
    local loFoxCode, ;
      luReturn
    loFoxCode = newobject(This.cProxyClass, ;
      This.cProxyClasslib)
    if vartype(loFoxCode) = 'O'
      luReturn = loFoxCode.Main(This.oFoxCode, ;
        'ConnectionMgr', 'Connectionmgr', ;
        'Path2\connmgr.vcx')
    else
      luReturn = ''
    endif vartype(loFoxCode) = 'O'
    return luReturn
  endproc
enddefine
```

This code defines a subclass of the FoxCodeScript class defined in the IntelliSense application specified by the _CODESENSE system variable. This subclass overrides the Main method, which is called by IntelliSense, to instantiate the FFIFoxCode class in FFI.VCX and call its Main method, passing it a reference to the IntelliSense data object (which contains information about what the user typed and other IntelliSense settings), the namespace the user is working on, and the class and library for that namespace. As a result of this script, FFIFoxCode.Main is called for all IntelliSense tasks for this namespace, such as when you select the namespace from the IntelliSense list displayed when you type LOCAL SomeVariable AS or when you type one of the "trigger" characters—such as a period, an opening parenthesis, or an equals sign—in a statement containing the name of the variable the class is instantiated into.

FFIFoxCode

The FFIFoxCode class does all of the custom IntelliSense work for FFI, so let's examine this class in detail.

The Init method just does two things: turns on debugging in system components (without this, you can't easily debug problems in the code) and opens the table a class and its members are registered in (FFI.DBF) by calling OpenFFITable. If the table can't be opened, Init displays an error message and returns .F. so the class isn't instantiated.

```
* Turn debugging on.

sys(2030, 1)

* Open the FFI ("Favorites for IntelliSense ") table.

local llReturn
llReturn = This.OpenFFITable()
if not llReturn
  messagebox('Could not open the Favorites for ' + ;
    'IntelliSense table.', 64, ;
    'IntelliSense Handler')
endif not llReturn
return llReturn
```

The Main method, called from the IntelliSense script, handles all of the IntelliSense tasks for FFI. As we saw earlier, the script passes a FoxCode object, the namespace for the class, and the class and library to Main. If the namespace is found in the MenuItem property of the FoxCode object, we must be on the LOCAL SomeVariable AS statement, so Main calls the HandleLOCAL method to deal with it. Otherwise, Main determines which character triggered IntelliSense and calls the GetFFIMember method to determine which member of the class you typed (it could also be the class itself) and return a SCATTER NAME object from the appropriate record in the FFI table. If the trigger character is a period, we need to display a list of the registered members of the class, so Main calls DisplayMembers to do the work. If the trigger character is an opening parenthesis, an equals sign, or a comma and there's code in the SCRIPT memo of

the FFI record, that code is executed. This script code could, for example, specify a list of enumerated values that IntelliSense should display for a property value or a parameter of a method (we'll see an example of that later). Finally, if the TIP memo of the FFI record is filled it, Main uses it as the tooltip for IntelliSense. This is usually used to display the signature of a method (for example, "Login(UserName as String, Password as String) as Boolean").

```
lparameters toFoxCode, ;
  tcNameSpace, ;
  tcClass, ;
  tcLibrary
local lcReturn, ;
  lcTrigger, ;
  loData
with toFoxCode

* If we're on the LOCAL statement, handle that by
* returning text we want inserted.

  lcReturn = ''
  if atc(tcNameSpace, .MenuItem) > 0
    lcReturn = This.HandleLOCAL(toFoxCode, ;
      tcNameSpace, tcClass, tcLibrary)

* Get the character that triggered IntelliSense and
* figure out which member the user typed.

  else
    lcTrigger = right(.FullLine, 1)
    loData    = This.GetFFIMember(.UserTyped, ;
      tcClass)
    do case

* We can't find the member in the FFI table, so do
* nothing.

      case vartype(loData) <> 'O'

* If we were triggered by a ".", display a list of
* members.

      case lcTrigger = '.'
        This.DisplayMembers(toFoxCode, loData)

* If we were triggered by a "(" (to start a method
* parameter list), an "=" (for a property), or ","
* (to enter a new parameter) and we have a script,
* execute it.

      case inlist(lcTrigger, '=', '(', ',') and ;
        not empty(loData.Script)
        lcReturn = execscript(loData.Script, ;
          toFoxCode, loData)

* If we were triggered by a "(" or ",", display the
* tooltip for the method (normally the method
* signature).

      case inlist(lcTrigger, '(', ',') and ;
        not empty(loData.Tip)
        .ValueTip  = loData.Tip
        .ValueType = 'T'
    endcase
  endif atc(tcNameSpace, .MenuItem) > 0
endwith
return lcReturn
```

We won't look at the code for HandleLOCAL here; feel free to examine this method yourself. Main calls it when you type LOCAL SomeVariable AS and choose one of the registered namespaces from the

type list. All it does is generate a NEWOBJECT() statement for the class so you don't have to. The only complication for this method is determining the case to use for NEWOBJECT(). (I could have hard-coded this as "newobject" since I always use lower-case for FoxPro keywords, but decided to be nice to other developers who may have different ideas about casing.) This is solved by looking in the IntelliSense table for a record with TYPE = "F" (for "function") and ABBREV = "NEWO" (the abbreviation for NEWOBJECT) and using the value in the CASE field.

GetFFIMember, called from Main, looks for the class or member you typed in the FFI table. It uses the UserTyped property of the FoxCode object (passed as the first parameter), which contains the text you typed pertaining to the namespace. For example, when you type "llStatus = ConnMgr.ExecuteSQLStatement(", UserTyped contains "ConnMgr.ExecuteSQLStatement." GetFFIMember starts by finding the record for the specified class in the FFI table. If a period is included in UserTyped, it then looks for the appropriate member record. If it finds the correct record, it returns a SCATTER NAME object from that record.

```
lparameters tcUserTyped, ;
  tcClass
local loReturn, ;
  lcUserTyped, ;
  llFound, ;
  lnPos, ;
  lcMember, ;
  lnSelect

* Grab what the user typed. If it ends with an
* opening parenthesis, strip that off.

loReturn    = .NULL.
lcUserTyped = alltrim(tcUserTyped)
if right(lcUserTyped, 1) = '('
  lcUserTyped = substr(lcUserTyped, ;
    len(lcUserTyped) - 1)
endif right(lcUserTyped, 1) = '('

* Find the record for the class in the FFI table. If
* there's a period in the typed text, try to find a
* record for the member.

if seek(upper(padr(tcClass, len(__FFI.CLASS))), ;
  '__FFI', 'CLASS')
  llFound = .T.
  lnPos   = at('.', lcUserTyped)
  if lnPos > 0
    lcMember = alltrim(__FFI.MEMBER) + ;
      substr(lcUserTyped, lnPos)
    llFound  = seek(upper(padr(lcMember, ;
      len(__FFI.MEMBER))), '__FFI', 'MEMBER')
  endif lnPos > 0

* If we found the desired record, create a SCATTER
* NAME object for it.

  if llFound
    lnSelect = select()
    select __FFI
    scatter memo name loReturn
    select (lnSelect)
  endif llFound
endif seek(upper(padr(tcClass ...
return loReturn
```

DisplayMembers is called from Main to tell IntelliSense to display a list of registered members for the class when you type a period in the command line. DisplayMembers calls GetMembers to retrieve a collection of members for the specified class (we won't look at that method here). It then fills the Items array of the FoxCode object with the names and descriptions of the members and sets the object's ValueType property to "L," which tells IntelliSense to display a listbox with the contents of the Items array.

This code shows one slight design flaw in IntelliSense: the FoxCode object has a single Icon property which contains the name of the image file to display in the listbox. What is actually needed is an additional column in the Items array, since in this case, we want to display different images for properties and methods. Unfortunately, we get only a single image displayed for all items.

```
lparameters toFoxCode, ;
  toData
local loMembers, ;
  lnI, ;
  loMember
with toFoxCode

* Get a collection of members for the current class.

  loMembers = This.GetMembers(alltrim(toData.Member))
  if loMembers.Count > 0

* Add each member to the Items array of the FoxCode
* object.

    dimension .Items[loMembers.Count, 2]
    for lnI = 1 to loMembers.Count
      loMember       = loMembers.Item(lnI)
      .Items[lnI, 1] = loMember.Name
      .Items[lnI, 2] = loMember.Description
      if loMember.Type = 'P'
        .Icon = home() + 'ffc\graphics\propty.bmp'
      else
        .Icon = home() + 'ffc\graphics\method.bmp'
      endif loMember.Type = 'P'
    next loMember

* Set the FoxCode object's ValueType property to "L",
* meaning display a listbox containing the items
* defined in the Items array.

    .ValueType = 'L'
  endif loMembers.Count > 0
endwith
```

Some advanced uses

There are some interesting things you can do with FFI to make it even easier to use. One is that classes can contain a hierarchy of objects, and by registering the classes contained as members of a class with FFI, you can get clean IntelliSense on those members. For example, the cApplication class instantiates an instance of the cUser class into its User property at runtime. Normally, you wouldn’t get IntelliSense on the User property because it doesn’t contain an object at design time. However, if you open the cUser class and specify “Application.User” as the namespace, you get IntelliSense on the User member of the Application namespace, as you can see in **Figure 5**.

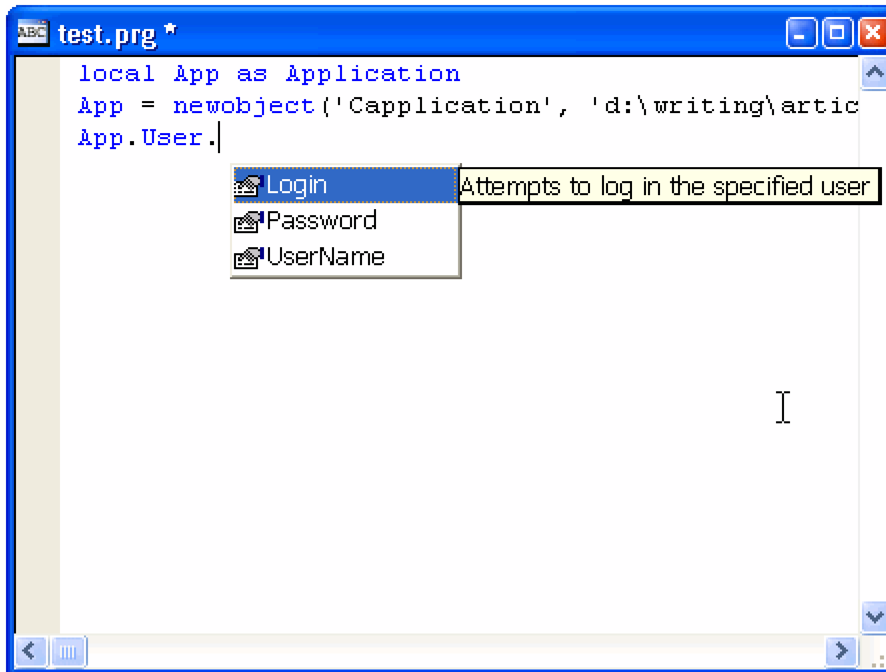

Figure 5. You can even get IntelliSense on objects instantiated at runtime if they're registered in a sub-namespace of the main namespace.

Another useful thing is to script how IntelliSense deals with a member by entering code in the Script editbox of the FFI builder form. For example, the DatabaseSpecifier property of ConnectionMgr indicates how the database should be specified in an ODBC connection string. For some databases, such as Access and dBase, you specify the database using "dbq=database path", while for others, you specify "database=database name." Rather than expecting the developer to know that the valid choices for DatabaseSpecifier are "database" and "dbq" (and to type them without any mistakes), you can tell IntelliSense to display a list of the valid values in a similar manner to the way the list of members is specified: by filling in the Items array of the FoxCode object and setting its ValueType property to "L." The following script code for DatabaseSpecifier does exactly that. Notice that all scripts are passed a reference to the FoxCode object and the SCATTER NAME object for the FFI record. Also note that quotes must be included in the values in column 1 if the values are strings, or else they won't be inserted into the code properly. Finally, note that the return value of the script is sent to IntelliSense, so unless you want that value inserted into the code, return an empty string and set ValueType to something other than "V."

```
lparameters toFoxCode, ;
  toData
dimension toFoxCode.Items[2, 2]
toFoxCode.Items[1, 1] = "'database'"
toFoxCode.Items[1, 2] = 'Used for most databases'
toFoxCode.Items[2, 1] = "'dbq'"
toFoxCode.Items[2, 2] = 'Used for Access and dBase'
toFoxCode.ValueType   = 'L'
return ''
```

Here's a script that displays a context-sensitive tooltip for some of the parameters of a method and a values list for one of them:

```
lparameters toFoxCode, ;
  toData
local lcMember, ;
  lnPos, ;
  lcParameters, ;
  lnParameters
lcMember     = alltrim(substr(toData.Member, ;
```

```
  rat('.', toData.Member) + 1))
lnPos        = atc(lcMember, toFoxCode.FullLine)
lcParameters = substr(toFoxCode.FullLine, ;
  lnPos + len(lcMember))
lnParameters = occurs(',', lcParameters) + 1
do case
  case lnParameters = 1
    toFoxCode.ValueTip  = 'First parameter tooltip'
    toFoxCode.ValueType = 'T'
  case lnParameters = 2
    toFoxCode.ValueTip  = 'Second parameter tooltip'
    toFoxCode.ValueType = 'T'
  case lnParameters = 3
    dimension toFoxCode.Items[2, 2]
    toFoxCode.Items[1, 1] = '1'
    toFoxCode.Items[1, 2] = 'Description for value 1'
    toFoxCode.Items[2, 1] = '2'
    toFoxCode.Items[2, 2] = 'Description for value 2'
    toFoxCode.ValueType = 'L'
endcase
```

Summary

By customizing the way IntelliSense works, we make the best productivity tool in VFP even more powerful. Using FFI, you can make working with frequently used classes (or even better, those that aren't frequently used so you're not as familiar with them) much easier because IntelliSense only displays those members you want to see, displays customized tooltips for each one, can provide IntelliSense on members instantiated at runtime, and can provide lists of values for enumerated properties or parameters of methods.

Thanks to Randy Brown, Program Manager for Visual FoxPro with Microsoft, for pointing me on the path to figuring out how to make IntelliSense only display the members you want.

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, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all 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 long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com