Using Assign methods

Assign methods let you take action when a property changes.

Tamar E. Granor, Ph.D.

In my last article, I introduced Access and Assign methods and showed some examples of Access methods. This time, I'll explore Assign methods, which fire when the corresponding property changes, essentially giving you a „property changed" event.

I've found more ways to use Assign methods than Access methods. Although an Assign method lets you change the value assigned, I rarely use that capability. More often, my Assign method code lets me ensure that additional things happen when the value is saved. Often, the goal is encapsulation; by using an Assign method, I keep the code that changes a property from having to know what a change in that property affects.

As with the Access method article and the two on BindEvent() that preceded it, some of the examples here are drawn from a sample Library application I created, which is included in this month's downloads.

Updating a timestamp

In one of my client applications, we track the current value (the "Actual Value") of each of a set of items. These values are read from actual hardware and we want to know not just the value, but when it was read. A business object contains the information for a single item; there's a collection of such items. The business object has a cActual property for the current value and a tLastRead property for the timestamp. An Assign method for cActual updates tLastRead, as in Listing 1.

Listing 1. The Assign method for a property that tracks values read from hardware updates the timestamp for the value.

```
LPARAMETERS tuNewValue

IF NOT (ALLTRIM(THIS.cActual) == ;
        ALLTRIM(m.tuNewValue))
   THIS.cActual = m.tuNewValue

   * Set last read time.
   THIS.tLastRead = DATETIME()
ENDIF
```

Set a "dirty" flag

The code in Listing 1 is actually only part of what we do when we read a new hardware value. We need a way to know whether the data has changed since it was last stored, that is, a "dirty" flag. Assign methods provide that, as well. We have an application property, lIsDirty. Whenever we save data or open a new data file, we clear that property. The Assign method (from Listing 1) includes the additional line of code shown in Listing 2 inside the IF; we use the same code in the Assign methods of all properties where a change indicates that the data is now different than it was at the last save.

Listing 2. One line of code in an Assign method (plus a little application-level code) lets you keep track of whether data has changed since it was last saved.

```
goApp.lIsDirty = .T.
```

In the Library application, we already have a lot of the framework in place for keeping an application-wide "dirty" flag. Each form has the lDataChanged property (described in my March, 2012 article). We can use those properties to determine whether there's any unsaved data in the application. To do so, we add an Assign method to lDataChanged in frmBase. That method calls an application object method to update the application-wide "dirty" flag.

Just to demonstrate another possibility, lDataChanged_Assign also updates the form caption so that whenever there's unsaved data, it includes an asterisk. (The VFP editor does the same thing.) The method is shown in Listing 3.

Listing 3. This code in the Assign method of the form's lDataChanged property aids in keeping an application-wide "dirty" flag, and has each form's Caption indicate whether it currently has unsaved changes.

```
LPARAMETERS tuNewValue
This.lDataChanged = tuNewValue

* Set app-level dirty flag
IF VARTYPE("goApp") = "O" AND ;
   NOT ISNULL(goApp) AND ;
   PEMSTATUS(goApp, "SetDirtyFlag", 5)
   goApp.SetDirtyFlag(m.tuNewValue)
ENDIF

* Update form caption
IF m.tuNewValue
   IF RIGHT(This.Caption, 1) <> "*"
      This.Caption = This.Caption + " *"
   ENDIF
ELSE
   IF RIGHT(This.Caption, 1) = "*"
      This.Caption = LEFT(This.Caption, ;
        LEN(This.Caption)-2)
   ENDIF
ENDIF
```

The application object's SetDirtyFlag method, along with an application object property, lUnsavedData, does the rest of the job. SetDirtyFlag, shown in Listing 4, checks the value passed to it; if it's .T., some form has unsaved data, so the flag is set to .T. If the parameter is .F., we know that at least one form has just either saved data or restored the previous data, but we don't know the state of the other forms, so we loop through them until we find one with unsaved data.

Listing 4. This method manages an application-wide "dirty" flag, so that we can know just by checking a single property whether there's any unsaved data.

```
PROCEDURE SetDirtyFlag(lNewValue)
* Set the application dirty flag. If the
* parameter is true, then some form has
* changed data, and we can just set this flag.
* If the parameter is false, some form has
* just saved or reverted its changed data
* and we need to look at all open forms.

IF m.lNewValue
   This.lUnsavedData = .T.
ELSE
   This.lUnsavedData = .F.
   FOR EACH oForm IN _VFP.Forms FOXOBJECT
      IF PEMSTATUS(oForm, "lDataChanged", 5)
         IF oForm.lDataChanged
            This.lUnsavedData = .T.
            * No need to find more than one
            EXIT
         ENDIF
      ENDIF
   ENDFOR
ENDIF

RETURN
```

In most data-entry applications, the next step would be to check the "dirty" flag on exit and prompt the user to save changes if there's unsaved data. The Library application was designed so that when you close a form, the data is automatically saved, so there's no need to prompt.

Delegate handling of a new value

For one application, I needed the ability to manage a complex set of user preferences. The preference items correlated more or less directly to either application object properties or properties of objects managed by the application object. For example, one preference addresses how often garbage collection takes place; this property needs to become the Interval for a timer. By creating an application object property for each preference and giving them Assign methods, I ensured that the appropriate updates happen automatically.

The Library application has only a couple of items in its Preferences form, shown in Figure 1. Use large toolbars maps directly to an application property; it uses BindEvent() to trigger resizing of the buttons in the toolbar.

Figure 1. The Library application's preferences form relies on Assign methods to ensure that the inactivity timer is set appropriately.

The checkbox that controls whether the application shuts down after a specified period of inactivity also maps onto an application property. However, the timer that tracks inactivity (described in my May, 2012 article) needs to be appropriately enabled or disabled when the user changes the checkbox. Similarly, the spinner that determines how long to wait for activity needs to set the Interval for that timer. (None of this happens until the user closes the Preferences form.) Both items use Assign methods to ensure that the appropriate changes occur.

When the user closes the form, the control values are stored to corresponding application properties. (The spinner value is multiplied by 60000, the number of milliseconds in a minute, first.) The adjusted spinner value is stored in an application property called nActivityTimerInterval. The nActivityTimerInterval_Assign method sets the timer's Interval, as shown in Listing 5. Note that we need to store the value in the application property, as well as setting the timer's Interval. Otherwise, the next time we open the Preferences dialog, it won't set the spinner to the current value.

Listing 5. This assign method sets the Interval for the activity timer when the user changes the setting in Preferences.

```
PROCEDURE nActivityTimerInterval_Assign( ;
  nNewValue)
* Something changed interval for activity
* timer. Propagate to timer

IF VARTYPE(m.nNewValue) = "N" AND ;
   m.nNewValue > 0 AND ;
   m.nNewValue <> This.oActivityTimer.Interval
    This.nActivityTimerInterval = m.nNewValue
    This.oActivityTimer.Interval = m.nNewValue
ENDIF

RETURN
```

The checkbox value is stored to the lTrackUserActivity property. Its Assign method enables or disables the timer, as shown in Listing 6.

Listing 6. When the user changes the preference for tracking user inactivity, the Assign method of lTrackUserActivity fires.

```
PROCEDURE lTrackUserActivity_Assign(lNewValue)
* Tracking decision changed. Set up or disable
* timer
```

```
IF VARTYPE(m.lNewValue) = "L" AND ;
   m.lNewValue <> This.lTrackUserActivity

   This.lTrackUserActivity = m.lNewValue
   IF This.lTrackUserActivity
      This.SetupActivityTimer()
   ELSE
      This.DisableActivityTimer()
   ENDIF
ENDIF

RETURN
```

Check for validity

One of the things you can do with an Assign method is check the new value of a property for validity and reject invalid values. For example, I wrote a Sudoku game in VFP a few years ago (as a demonstration of business objects). The bizGame object, which represents the game as a whole, has a property named nSize that holds the board size (the number of cells in either direction). Since Sudoku requires the game size to be a perfect square, nSize has an Assign method that checks; it's shown in Listing 7.

Listing 7. The nSize_Assign method of the bizGame object of a Sudoku application ensures that the specified game size is valid.

```
* Ensure that size is a perfect square
LPARAMETERS tuNewValue

IF SQRT(m.tuNewValue) = ;
   INT(SQRT(m.tuNewValue))
   This.nSize = tuNewValue
ENDIF

RETURN
```

Propagating data inside a container

One of the main ways I use Assign methods is to push data down a hierarchy inside a container, so that only the container is exposed to the world. There are a variety of behaviors along these lines that are useful.

In my last article, I showed how to use Access methods to provide dynamic tooltips and to have the same tooltip for a container and its contents. Assign provides an alternate way to do the latter, in situations where you don't need dynamic tips. That is, this approach works when you want to assign a fixed tooltip to a container and have all the controls inside show the same tip.

This code uses a custom logical property, lPropagateTooltipsDown, in the base container class. The ToolTipText_Assign method contains the code in Listing 8. If the container has the flag set to .T., when the ToolTip changes, we loop through the objects in the container and change ToolTipText for each of them.

Listing 8. This code in the ToolTipText_Assign method lets you propagate a container's ToolTipText to the contained objects.

```
LPARAMETERS tuNewValue
This.ToolTipText = tuNewValue

LOCAL oObject
IF THIS.lPropagateToolTipDown
   FOR EACH oObject IN THIS.OBJECTS FOXOBJECT
      IF PEMSTATUS(m.oObject, ;
                   "ToolTipText", 5)
         * don't use ToolTipText on the right
         * in case there's an access method
       m.oObject.ToolTipText = m.tuNewValue
      ENDIF
   ENDF
ENDIF
```

Adjust a label

In one application, I needed to have a label running vertically inside a shape. In the relevant form (shown in Figure 2.), the label's captions are determined dynamically based on the data. I created a label subclass and gave Caption an Assign method. That method calls a custom TurnCaption method to rebuild the caption adding the appropriate punctuation to make it display vertically. For the double-wide shapes in the top row, the same method breaks the caption into two strings of about the same length and builds the appropriate caption string. TurnCaption is shown in Listing 9.

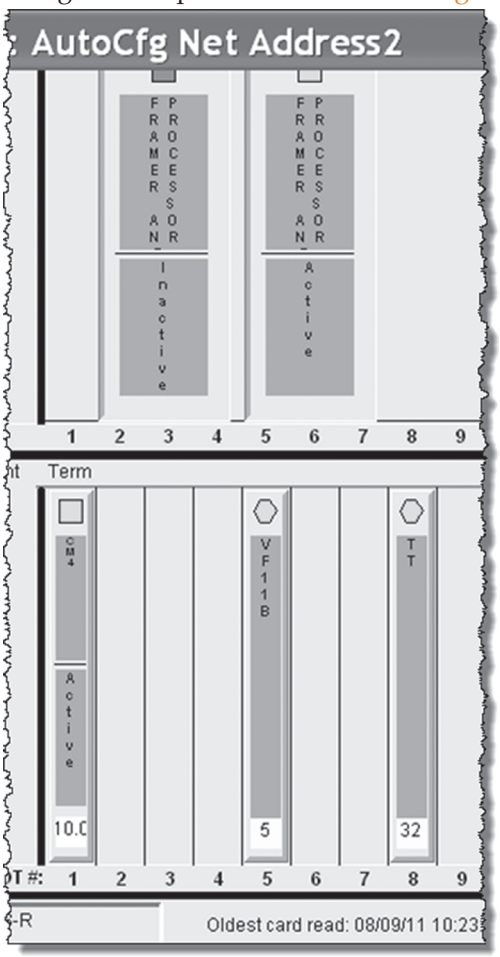

Figure 2. The vertical labels inside the boxes here are set up by an Assign method.

Listing 9. This code is called by the Caption_Assign method of the vertical labels shown in Figure 2. The vertical labels inside the boxes here are set up by an Assign method..

```
LPARAMETERS cCaption, lTwoColumns

#DEFINE LF CHR(10)
LOCAL cTurnedCaption, nLength, nChar

m.cCaption = CHRTRAN(CHRTRAN(m.cCaption, ;
  CHR(160), " "), LF, "")
nLength = LEN(m.cCaption)

IF m.lTwoColumns
   * Find a dividing point
   nWords = GETWORDCOUNT(m.cCaption)
   IF m.nWords > 1
      * Break on a word break
      cColumn1 = GETWORDNUM(m.cCaption, 1)
      cColumn2 = ALLTRIM(SUBSTR(m.cCaption, ;
        LEN(m.cColumn1) + 1))
      cNextWord = GETWORDNUM(m.cColumn2, 1)
      nWord = 2

      DO WHILE m.nWord <= m.nWords-1 AND ;
         LEN(m.cColumn1) + LEN(m.cNextWord) <;
         LEN(m.cColumn2) - LEN(m.cNextWord)
            cColumn1 = m.cColumn1 + " " + ;
              m.cNextWord
         cColumn2 = ALLTRIM( ;
           SUBSTR(m.cColumn2, ;
           LEN(m.cNextWord) + 1))
         cNextWord = GETWORDNUM(m.cColumn2, 1)
         nWord = nWord + 1
      ENDDO

      nLength = MAX(LEN(m.cColumn1), ;
        LEN(m.cColumn2))

   ELSE
      * Just break it in half
      cColumn1 = LEFT(m.cCaption, ;
        FLOOR(m.nLength/2))
      cColumn2 = RIGHT(m.cCaption, ;
        CEILING(m.nLength/2))

      * Pad shorter string
      cColumn1 = PADR(m.cColumn1, ;
        LEN(m.cColumn2))
   ENDIF
ELSE
   cColumn1 = m.cCaption
   cColumn2 = ""
ENDIF

cTurnedCaption = ""

FOR nChar = 1 TO nLength
   cTurnedCaption = m.cTurnedCaption + ;
     EVL(SUBSTR(m.cColumn1, nChar, ;
         1),CHR(160))
   IF m.lTwoColumns
      cTurnedCaption = m.cTurnedCaption + ;
        CHR(160) + CHR(160) + ;
        SUBSTR(m.cColumn2, nChar, 1)
   ENDIF
   cTurnedCaption = m.cTurnedCaption + LF
ENDFOR

RETURN m.cTurnedCaption
```

For the Library application, you could use this code if you want to display books graphically with titles running down the spine.

The Bottom Line

What BindEvent(), Access methods and Assign methods all have in common is that they give you more opportunities to „set it and forget it." That allows you to create classes with complex behavior that you can control by setting a property or two.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of about a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer. *Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*