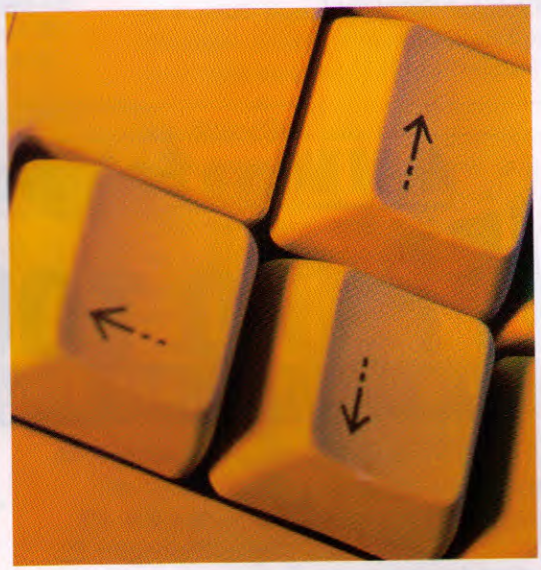
Add QuickFill to Your Combos

Subclass combos to make them work like the ones in Quicken.

By Tamar E. Granor, Editor

When I speak about user interfaces, I often ask people to name their favorite applications. People mention various games but, invariably, someone brings up Intuit's Quicken.

Why do people love an application that keeps track of their checkbook? Because it has some great user interface features that make it a pleasure to use. Probably the best known of those features is QuickFill, which cuts down dramatically on the amount of typing a user has to do.



What is QuickFill?

Ordinary combo boxes are good for looking up data from a list. When the user is restricted to the listed items, combos are easy to use—just start typing, and the first matching entry appears. Keep typing, and you home in on the right choice. (This feature is called *incremental search* and can be turned off). In its drop-down list style, combos are good for both dialogs and heads-down data entry of things like cities and states. But when you use the drop-down combo style to let the user enter items not on the list as well as choose from the list, it gets messy. To get incremental search, the user has to open the combo before starting to type. Typing in a closed combo doesn't match the items on the list, unless the entire item is typed.

QuickFill offers the best of both worlds. As the user types into the combo, whether open or closed, not only does incremental search kick in, but the text portion of the combo shows the entire matching string—the portion that's being guessed is highlighted. Figure 1 shows QuickFill at work. The user has typed "Fr" and QuickFill matches it with "Frankenversand" (the form uses the Customer table from the TasTrade sample) highlighting "ankenversand" to indicate that it's a guess.

The July 1995 issue of FoxPro Advisor showed how to add QuickFill in FoxPro 2.x. In Visual FoxPro, it's much easier because we have access to many more events. This article shows how to create QuickFill combo boxes.

PRODUCTS

Visual FoxPro 6.0/5.0/3.0

Combo basics

Before we look at implementing QuickFill, a little background on the internals of combo boxes is useful. (For more information, see my article,

Tamar E. Granor, Ph.D., is editor of FoxPro Advisor magazine and co-author of the FoxPro Advisor Ask ADVISOR column. Tamar is also co-author (with Ted Roche) of the *Hackers Guide to Visual FoxPro 3.0 for Windows* (Addison-Wesley) and is hard at work on a new book, due this summer from Hentzenwerke Publishing. She's been a speaker at numerous FoxPro conferences. Tamar is an independent consultant specializing in applications using Microsoft FoxPro. tamar_granor@compuserve.com.

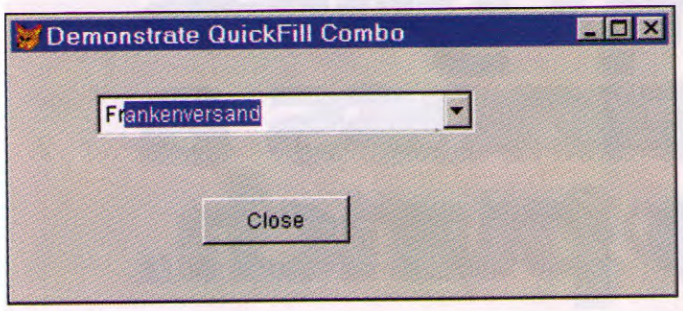


Figure 1: **QuickFill combo**—Like Quicken, you can type in just enough characters to identify the item you want.

"Smarter List and Combo Boxes," in the June 1996 issue of FoxPro Advisor.)

Combos use two different properties to store information about the chosen item. The Value property contains either the text of the chosen item or its position in the list (depending on whether Value is character or numeric and, in Visual FoxPro 6.0 and 5.0, on the setting of the BoundTo property). However, when the user types in an item not on the list, Value is left empty and the user's entry is stored in DisplayValue. (In Visual FoxPro 5.0 and higher, the entry is also stored, unformatted, in Text.)

The data in combos can be drawn from a number of different sources. The RowSourceType property determines the type of source and RowSource identifies the actual source. The QuickFill combos in this article work with RowSourceTypes 0 (None), 1 (Value), 2 (Alias), 5 (Array), and 6 (Fields).

Combos have several properties that deal with the contents of the list and the chosen item. The List property is an array that contains the actual items in the list (regardless of RowSourceType). ListIndex indicates the position of the currently selected item in the List array.

The game plan

To create a QuickFill combo, we can subclass Visual FoxPro's base combo box class. The new class is called cboQFill.

The basic strategy is to grab each keystroke typed into the combo and see how it affects the entry so far. The combo's KeyPress method lets us see each keystroke before it's actually sent to the combo—we can hijack the user's entry there and handle it as we want. KeyPress is the control center for the whole QuickFill technique. A custom method called HandleKey actually looks at which key was pressed and figures out what to do with it.

When the user types an alphanumeric or punctuation character, a backspace or a space, we need to search the list to find the first match. A custom method, Search, does this.

Unfortunately, Visual FoxPro's built-in array functions don't work on built-in array properties, so we can't just apply ASCAN() to List. We could search List sequentially, but that could be slow on large lists. When RowSourceType is array, we can apply ASCAN() to the original array. Similarly, for a combo based on a table, SEEK is the fastest way to find a matching entry.

Rather than put a complex CASE statement in the search method, a better choice is to leave Search empty at the top of our class hierarchy and create separate subclasses for each kind of RowSourceType, filling in the Search method at that level. Three subclasses are presented here:

- cboListQFill, for list-based combos (RowSourceTypes 0 and 1).
- cboArrQFill, for array-based combos (RowSourceType 5).
- cboAlsQFill for table-based combos (RowSourceTypes 2 and 6).

All four classes can be found in the Combos class library on this issue's Professional Resource CD.

Navigation

Navigation keys do various things in QuickFill combos. The right arrow and End keys move the cursor to the end of the current value, turning off highlighting. The left arrow also turns off the highlight and positions the cursor between the last two characters of the entry. Home turns off highlighting and moves to the beginning of the entry. In Quicken, Backspace removes the last character typed and leaves no guesswork. Since I've always found that behavior annoying, my QuickFill combo removes the last character and searches the list for the first match to the remaining string. An include file (KEYS.H) contains the codes for the navigation keys to make method code more readable. Here's the contents of KEYS.H:

```
* Include file for special keystroke.
* Used by QuickFill combo.

#DEFINE KEY_BACKSPACE       127
#DEFINE KEY_DEL               7
#DEFINE KEY_RTARROW           4
#DEFINE KEY_LTARROW          19
#DEFINE KEY_HOME              1
#DEFINE KEY_END               6
#DEFINE KEY_SHIFT_LTARROW    52
#DEFINE KEY_SHIFT_RTARROW    54
#DEFINE KEY_CTRL_LTARROW     26
#DEFINE KEY_CTRL_RTARROW      2
#DEFINE KEY_CTRL_HOME        31
#DEFINE KEY_CTRL_END         23

* Add codes for CtrlAltShift status
#DEFINE KEY_MODIFIER_NONE          0
#DEFINE KEY_MODIFIER_SHIFT         1
#DEFINE KEY_MODIFIER_CTRL          2
#DEFINE KEY_MODIFIER_CTRLSHIFT     3
#DEFINE KEY_MODIFIER_ALT           4
#DEFINE KEY_MODIFIER_ALTSHIFT      5
#DEFINE KEY_MODIFIER_ATLCTRL       6
#DEFINE KEY_MODIFIER_ALTCTRLSHIFT  7
```

Custom properties and methods

Several custom properties are used to allow communication between the various methods involved.

nLastKey—the key pressed, stored in KeyPress for use in HandleKey.

nLastModifier—any modifiers (Ctrl, Alt, Shift) of the key, stored in KeyPress for use in HandleKey.

lHandled—set in HandleKey to indicate whether the keystroke was dealt with there or should be added to the keyboard buffer and processed normally.

lSearch—set in HandleKey to indicate whether the Search procedure needs to be called.

nAnchor—stores the position at which highlighting begins. This lets the user do his own highlighting using the usual key combinations.

All the custom properties are protected to make them available only within this class and its subclasses.

The class has three custom methods, HandleKey and Search described above, and zReadMe that is the documentation for the class. zReadMe is public while the other two are protected.

Key handling code

As noted above, handling of keystrokes is split between two methods. KeyPress is an event method that is called whenever the user presses a key in the control. For QuickFill combos, it's the control center and contains this code:

```
LPARAMETERS nKeyCode, nShiftAltCtrl

* Grab the keycode to figure out what to do with it
THIS.nLastKey = nKeyCode
THIS.nLastModifier = nShiftAltCtrl
THIS.HandleKey()

IF THIS.lHandled
   NODEFAULT
   IF THIS.lSearch
      THIS.Search()
   ENDIF
ENDIF
```

This code saves the keystroke information to properties of the combo, then calls HandleKey to figure out what to do. HandleKey sets two properties, lHandled (to indicate whether the keystroke should be passed along to the combo) and lSearch (to indicate whether we have a new partial string and need to search for a match).

If HandleKey indicates that it processed the keystroke, NODEFAULT prevents the keystroke from being passed on to the combo. Then, if necessary, the Search method is called.

Most of the work occurs in HandleKey:

```
LOCAL cSoFar, nHoldLength

THIS.lHandled = .F.

DO CASE
CASE BETWEEN(THIS.nLastKey,65,90) OR ;
     BETWEEN(THIS.nLastKey,97,122) OR ;
     (BETWEEN(THIS.nLastKey,32,126) AND ;
      THIS.nLastModifier = KEY_MODIFIER_NONE)
   * Alphanumeric and punctuation
   cSoFar = LEFT(THIS.DisplayValue,THIS.SelStart)
   THIS.DisplayValue = PADR(cSoFar,THIS.SelStart) + ;
                       CHR(THIS.nLastKey)
   THIS.SelStart = LEN(TRIM(THIS.DisplayValue))
   * handle embedded blanks
   IF THIS.nLastKey = 32
      THIS.SelStart = THIS.SelStart+1
   ENDIF
   THIS.lSearch = .T.
   THIS.lHandled = .T.
   THIS.nAnchor = THIS.SelStart

ASE THIS.nLastKey = KEY_BACKSPACE
   * On backspace, remove one character from current position
   * and search
   IF THIS.SelStart<>0 AND ;
      THIS.SelStart+THIS.SelLength = ;
         LEN(TRIM(THIS.DisplayValue))
      cSoFar = LEFT(THIS.DisplayValue,THIS.SelStart)
      THIS.DisplayValue = LEFT(cSoFar,LEN(cSoFar)-1)
      THIS.SelStart = LEN(TRIM(THIS.DisplayValue))
      THIS.lSearch = .T.
      THIS.lHandled = .T.
      THIS.nAnchor = THIS.SelStart
   ENDIF

SE (THIS.nLastKey = KEY_RTARROW OR ;
      THIS.nLastKey = KEY_LTARROW OR ;
      THIS.nLastKey = KEY_END)
   * Move to end. For left arrow, we need
   * to move one to the left.
   IF THIS.SelStart+THIS.SelLength = ;
      LEN(TRIM(THIS.DisplayValue)) ;
      AND THIS.SelLength<>0
      THIS.SelStart = LEN(TRIM(THIS.DisplayValue))
      IF THIS.nLastKey = KEY_LTARROW
         THIS.SelStart = THIS.SelStart-1
      ENDIF
      THIS.SelLength = 0
      THIS.lSearch =.F.
   ELSE
      * Nothing is highlighted, so treat as ordinary
      * nav keys in the text box
      DO CASE
      CASE THIS.nLastKey = KEY_RTARROW
         IF THIS.SelStart <> LEN(TRIM(THIS.DisplayValue))
            THIS.SelStart = THIS.SelStart+1
         ENDIF
      CASE THIS.nLastKey = KEY_LTARROW
         IF THIS.SelStart <> 0
            THIS.SelStart = THIS.SelStart - 1
         ENDIF
      CASE THIS.nLastKey = KEY_END
         THIS.SelStart = LEN(TRIM(THIS.DisplayValue))
      ENDCASE
   ENDIF
   THIS.lHandled = .T.
   THIS.nAnchor = THIS.SelStart

CASE THIS.nLastKey = KEY_HOME
   * Move to beginning of text
   THIS.SelStart = 0
   THIS.nAnchor = 0
   THIS.SelLength = 0
   THIS.lSearch =.F.
   THIS.lHandled = .T.
```

Continued

```
CASE THIS.nLastKey = KEY_SHIFT_RTARROW AND ;
     THIS.nLastModifier <> KEY_MODIFIER_NONE
   * Shift-rightarrow does nothing if the highlight
   * goes to the end. Otherwise, it increases the
   * highlight to the right.
   IF THIS.SelStart+THIS.SelLength <> LEN(TRIM(THIS.Text))
      IF THIS.nAnchor = THIS.SelStart
         THIS.SelLength = THIS.SelLength+1
      ELSE
         nHoldLength = THIS.SelLength
         THIS.SelStart = THIS.SelStart+1
         THIS.SelLength = nHoldLength-1
      ENDIF

   ENDIF
   THIS.lHandled = .T.
   THIS.lSearch = .F.

CASE THIS.nLastKey = KEY_SHIFT_LTARROW AND ;
     THIS.nLastModifier <> KEY_MODIFIER_NONE
   * Shift-leftarrow does nothing if the highlight
   * starts at the beginning. Otherwise, it increases
   * the highlight to the left.
   IF THIS.SelStart <> 0
      IF THIS.nAnchor = THIS.SelStart + THIS.SelLength
         nHoldLength = THIS.SelLength
         THIS.SelStart = THIS.SelStart-1
         THIS.SelLength = nHoldLength+1
      ELSE
         THIS.SelLength = THIS.SelLength-1
      ENDIF

   ENDIF
   THIS.lHandled = .T.
   THIS.lSearch = .F.

OTHERWISE
   THIS.lSearch =.F.

ENDCASE

RETURN
```

HandleKey first determines the type of keystroke. Alphanumeric and punctuation keys are added to the accumulated string and mean that a search is needed (so lSearch is set to .T.). Notice that we can't just grab the combo's DisplayValue and add the new key to it. First, we must remove the guessed characters (the ones that are highlighted). We also set SelStart to indicate where highlighting is to begin if a match is found.

If the user clicked on BackSpace, we remove the last character actually typed (again, we have to strip off the guessed characters), then trigger a search.

Handling of the various navigation keys differs depending on whether anything is highlighted. If so, we use the QuickFill definitions; if not, they're treated as ordinary navigation keys.

Finally, any other keys are simply passed along to the combo to be handled normally.

List-based combos

Searching is handled by cboQFill's subclasses so that each can use the quickest approach. Things are simplest for cboListQFill, which does a sequential search of the combo's List property. Here's the Search method for this subclass:

```
* Search the list for a match to the keys so far
LOCAL cSoFar, nPos, cOldExact

cOldExact = SET("EXACT")
SET EXACT OFF

cSoFar = PADR(THIS.DisplayValue,THIS.SelStart)

IF LEN(cSoFar)=0
   THIS.ListIndex = 0
ELSE
   * Search through the list to find the first
   * match, starting from the current position
   FOR nPos = THIS.ListIndex TO THIS.ListCount
      IF UPPER(THIS.List[nPos]) = UPPER(cSoFar)
         THIS.ListIndex = nPos
         THIS.SelStart = LEN(cSoFar)
         THIS.SelLength = ;
            LEN(TRIM(THIS.DisplayValue))-LEN(cSoFar)
         EXIT
      ENDIF
   ENDFOR
ENDIF

SET EXACT &cOldExact
```

Notice that the comparison is case-insensitive. This is the way Quicken behaves and it makes things easy for users. Also, note that EXACT is set OFF at the beginning of the code. The whole idea of QuickFill is based on partial matches. (Of course, EXACT is reset to its original value on the way out.)

Since the QuickFill results make more sense when the list of items is sorted, the Sorted property is set to .T. in this subclass. It can be turned off, but the search then uses the natural order of the items.

Array-based combos

Array-based combos need somewhere to keep the array. CboArrQFill has a custom aContents property, which is an array. Its RowSource is set to THIS.aContents.

Searching in an array is fairly simple with the ASCAN() function. However, ASCAN() is case-sensitive and there's no way to tell it to ignore case. To maintain the speed benefits of ASCAN(), we keep another copy of the original array hidden inside the combo, with all items in upper case. This property is called aUpperContents—it's protected since there's no reason for any other objects to know about it.

The aUpperContents array needs to be filled in whenever the aContents array changes. The Requery method is the key here—it needs to be called whenever the array contents change to update the combo. So, Requery contains code to update aUpperContents:

```
* Make an upper case copy of the list for the combo

LOCAL nCnt

DIMENSION THIS.aUpperContents[1]
ACOPY(THIS.aContents,THIS.aUpperContents)

FOR nCnt = 1 TO ALEN(THIS.aUpperContents)
   THIS.aUpperContents[nCnt] = ;
      UPPER(THIS.aUpperContents[nCnt])
ENDFOR
```

Once you have the upper case array, searching is simple. Here's the Search method:

```
* Search the source array for a match to the string so far.

local cSoFar, nPos, cOldExact

cOldExact=SET("EXACT")
SET EXACT OFF

cSoFar = PADR(THIS.DisplayValue,THIS.SelStart)

IF LEN(cSoFar) = 0
   THIS.ListIndex = 0
ELSE
   nPos = ASCAN(THIS.aUpperContents,UPPER(cSoFar))
```

Continued

```
   IF nPos<>0
      THIS.ListIndex = ;
        ASUBSCRIPT(THIS.aUpperContents,nPos,1)
      THIS.SelStart = LEN(cSoFar)
      THIS.SelLength = ;
         LEN(TRIM(THIS.DisplayValue))-LEN(cSoFar)
   ENDIF
ENDIF

SET EXACT &cOldExact
```

As with list-based combos, things make more sense to the user if the array is sorted, but it isn't a requirement.

Table-based combos

The final subclass works with combos based on tables, with RowSourceType either 2-Alias or 6-Fields. These two types are slightly different. The Alias type has a RowSource that is simply the alias of the relevant table, and displays one or more columns from the beginning of that table. For the Fields type, RowSource contains a list of fields to be included. (For this class to work, the first item in the field list must contain the alias of the relevant table and the table must be open by the time the combo's Init runs.)

Three custom properties are needed to simplify searching:

cAlias—the alias for the table being searched.

cOrder—the order for the table being searched.

cField—the first field in the combo, which is the field to be searched.

All this information is available from the combo itself, so these properties don't have to be filled in by the developer. Instead, the Init method calls a custom SetAlias method that examines the RowSource and fills those properties. Here's SetAlias:

```
* Set the cAlias property by parsing RowSource

* User shouldn't set this one, so clear it out initially
THIS.cAlias = ""

DO CASE
CASE THIS.RowSourceType = 2
   * If RowSourceType is Alias, just grab the alias
   THIS.cAlias = THIS.RowSource
   THIS.cField = FIELD(1,THIS.cAlias)

CASE THIS.RowSourceType = 6
   * If RowSourceType is Fields, need to parse off the alias
   LOCAL nDotPos
   nDotPos = AT(".",THIS.RowSource)
   IF nDotPos<>0
      THIS.cAlias = LEFT(THIS.RowSource,nDotPos-1)
   ENDIF

   * In this case, we also need to know which field to search
   LOCAL nCommaPos
   nCommaPos = AT(",",THIS.RowSource)
   IF nCommaPos = 0
      nCommaPos = LEN(THIS.RowSource)-1
   ENDIF
   THIS.cField = SUBSTR(THIS.RowSource,nDotPos+1,nCommaPos-1)
ENDCASE

* Now set order to use.

IF USED(THIS.cAlias)
   THIS.cOrder=ORDER(THIS.cAlias)
ENDIF
```

If RowSourceType is 2, figuring out the alias is as simple as copying RowSource. In this case, cField is the first field of the table. For RowSourceType 6, the alias can be parsed out of the RowSource—it's whatever appears before a dot ("."). cField is the field that appears after that dot.

After we know what table we're looking at, we can simply ask it what its default order is with the ORDER() function. The assumption here is that the table is initially set to the correct order. However, SetAlias can be called explicitly anytime the RowSource or Order is changed. There's also an assumption here that the specified tag applies the UPPER() function (or that the field in question is always upper-case). If the tag doesn't use UPPER(), the Search method won't find a match when the user types in lower-case.

With these properties set, the Search code is straightforward. If cOrder is not empty, SEEK is used based on that tag. If no order is set for the table, LOCATE is used on the field indicated by cField. Here's the Search method:

```
* Search the source array for a match to the string so far.

local cSoFar, nPos, cOldExact

cOldExact=SET("EXACT")
SET EXACT OFF

cSoFar = PADR(THIS.DisplayValue,THIS.SelStart)

IF LEN(cSoFar) = 0
   THIS.ListIndex = 0
ELSE
   * Move to the right workarea
   IF NOT EMPTY(THIS.cAlias)
      LOCAL nOldSelect
      nOldSelect = SELECT()
      SELECT (THIS.cAlias)

      * Do we have an index?
      IF NOT EMPTY(THIS.cOrder)
         LOCAL cOldOrder
         cOldOrder = ORDER()
         SET ORDER TO (THIS.cOrder)

         * Now we can use seek for a fast search
         SEEK UPPER(cSoFar)

         SET ORDER TO (cOldOrder)
      ELSE
         * sequential search is required
         LOCATE FOR UPPER(EVAL(THIS.cField)) = ;
           UPPER(cSoFar)
      ENDIF
      IF FOUND()
         THIS.DisplayValue = EVAL(THIS.cField)
         THIS.SelStart = LEN(cSoFar)
         THIS.SelLength = ;
            LEN(TRIM(THIS.DisplayValue))-LEN(TRIM(cSoFar))
      ENDIF

      * Reset workarea
      SELECT (nOldSelect)
   ENDIF
ENDIF

SET EXACT &cOldExact
```

Using QuickFill combos

Putting these classes to work isn't difficult. Just drop one on a form and set RowSourceType and RowSource. For cboArrQFill, you also need to add some code to fill the aContents array. Be sure to call the Requery method afterward. For cboAlsQFill, be sure that the source table is open before the combo's Init.

After you build the classes (which are all found on this issue's PROFESSIONAL RESOURCE CD), you have a tool you can use over and over to keep your users as happy as Quicken's. ■