TBrowse and TBColumn basics

Data Based Advisor
April 01, 1993 | Gutierrez, Dan D.

This month's **column** applies the general "object-based" concepts defined last month by showing how to use two Clipper predefined classes: TBrowse and TBColumn. These classes have made **Clipper**'s previous database browsing technology, DBEDIT(), obsolete. DBEDIT() is a built-in function that can still be use, but there's little reason to do so. (In the manual it's listed as a Summer '87 compatibility function.) **Clipper**'s current data browsing facilities far surpass anything previously available in this language or any other xBase dialect. The key is the object-oriented direction of the browsing technology in **Clipper** 5.

We'll use the term TBrowse to encompass **Clipper**'s browsing technology and it should be understood that this includes both the TBrowse and TBColumn classes. Moreover, TBrowse can be used to browse not just information coming from a database, but any tabular data as well. For instance, arrays and DOS text files may be browsed via TBrowse.

The TBrowse and TBColumn classes

To browse tabular data in **Clipper**, you must use the TBrowse and TBColumn predefined classes. We use the TBrowse class to create and manipulate objects used as the primary mechanism for browsing table oriented data.

A TBrowse object depends on one or more TBColumn objects. The TBColumn class is used to create and manipulate objects used with TBrowse objects. A TBColumn object contains all the specifications required to define a single **column** of a TBrowse object.

Both classes have constructor functions used to create an instance of each class, also known as an "object." For TBrowse, there are the TBrowseNew() and TBrowseDB() constructors, and for TBColumn there is TBColumnNew().

In addition, there's a set of exported instance variables for both classes. Table 1 summarizes these variables. Think of the word "exported" to mean visible to the programmer to assign new values or access the value of the variable. In true object-oriented programming (OOP) languages where you can create classes, non-exported instance variables become possible. In this case, the variables are used only internally by an object.

Table - 1 : Exported Instance Variables of TBrowse and TBColumn Classes

TBrowse -- Exported Instance Variable	
AutoLite	Logical value to control highlighting
Cargo	User-definable variable
ColCount	Number of browse columns
ColorSpec	Color table for the TBrowse display
ColPos	Current cursor column position
ColSep	Column separator character
FootSep	Footing separator character
Freeze	Number of columns to freeze
GoBottomBlock	Code block executed by TBrowse:GoBottom()
GoTopBlock	Code block executed by TBrowse:GoTop()
HeadSep	Heading separator character
HitBottom	Indicates the end of available data
HitTop	Indicates the beginning of available data
LeftVisible	Indicates position of leftmost unfrozen column in display
nBottom	Bottom row number for the TBrowse display
nLeft	Leftmost column for the TBrowse display
nRight	Rightmost column for the TBrowse display
nTop	Top row number for the TBrowse display
RightVisible	Indicates position of rightmost unfrozen column in display
RowCount	Number of visible data rows in the TBrowse display
RowPos	Current cursor row position
SkipBlock	Code block used to reposition data source
Stable	Indicates if the TBrowse object is stable

TBColumn -- Exported Instance Variables	
Block	Code block to retrieve data for the column
Cargo	User-definable variable
ColorBlock	Code block that determines color of data items
ColSep	Column separator character
DefColor	Array of numeric indexes into the color table
Footing	Column Footing
FootSep	Footing separator character
Heading	Column heading
HeadSep	Heading separator character
Picture	Column picture string
Width	Column display width

In the case of TBrowse, there are also several method functions available. (TBColumn doesn't have any method functions). The TBrowse method functions are shown in Table 2, along with a description of their purpose and return value.

Table -2 Exported and Miscellaneous Methods of TBrowse Class

TBrowse -- Exported Methods	
Cursor Movement Methods	
Down()	Moves the cursor down one row
end()	Moves the cursor to the rightmost visible data column
GoBottom()	Repositions the data source to the bottom of file
GoTop()	Repositions the data source to the top of file
Home()	Moves the cursor to the leftmost visible data column
Left()	Moves the cursor left one column
PageDown()	Repositions the data source downward
PageUp()	Repositions the data source upward
PanEnd()	Moves the cursor to the rightmost data column
PanHome()	Moves the cursor to the leftmost visible data column
PanLeft()	Pans left without changing the cursor position
PanRight()	Pans right without changing the cursor position
Right()	Moves the cursor right one column
Up()	Moves the cursor up one row

TBrowse -- Miscellaneous Methods	
AddColumn()	Adds a TBColumn object to the TBrowse object
ColorRect()	Alters the color of a rectangular group of cells
ColWidth()	Returns the display width of a particular column
Configure()	Reconfigures the internal settings of the TBrowse object
DeHilite()	Dehighlights the current cell
DelColumn()	Delete a column object from a browse
ForceStable()	Performs a full stabilization
GetColumn()	Gets a specific TBColumn object
Hilite()	Highlights the current cell
InsColumn()	Insert a column object in a browse
Invalidate()	Forces redraw during next stabilization
RefreshAll()	Causes all data to be refreshed during the next stabilize
RefreshCurrent()	Causes the current row to be refreshed on next stabilize
SetColumn()	Replaces one TBColumn object with another
Stabilize()	Performs incremental stabilization

TBrowse requirements

Admittedly, generating a database browsing routine in **Clipper** takes a bit of effort, but it really is simple if you break the process down into component parts. There are four basic steps in programming a TBrowse:

1. Create a TBrowse object using either the TBrowseNew() or TBrowseDB() constructors

2. Create multiple TBColumn objects, one for each **column** in the browse using the TBColumnNew() constructor

3. Inform the TBrowse object of the columns in the browse by sending the addColumn() message for each TBColumn object previously created.

4. Set up the primary browse loop to display the data and handle keystroke exceptions.

For any browse to take place, you must first create at least one TBrowse object and one TBColumn object. The TBrowse object controls the browse in general and each TBColumn object is responsible for controlling a particular **column** of the display. A browse normally consists of a single TBrowse object and several TBColumn objects. To illustrate this concept, consider the following structure for EMPLOYEE.DBF:

LAST	C	20	0
FIRST	C	15	0
MI	C	1	0
ADDR1	C	25	0
ADDR2	C	25	0
CITY	C	15	0
STATE	C	2	0
ZIP	C	10	0
HOME_PH	C	13	0
WORK_EXT	C	5	0
HIRE_DATE	D	8	0
MO_SAL	N	7	2
FULL_PART	L	1	0

Let's go through the four steps to browse this database.

Step 1: Create the TBrowse object.

Say we want to establish a browse for only the LAST, FIRST, HIRE_DATE, and MO_SAL. The code required to create a TBrowse object would be:

```
LOCAL oEmpBrowse, oColumn1, oColumn2,
      oColumn3, oColumn4
oEmpBrowse := TBrowseDB ( 5, 10, 20, 70 )
```

The TBrowseDB() constructor function builds an instance (such as an object) of the TBrowse class and assigns it to a memory variable oEmpBrowse. We could have used the other TBrowse constructor, TBrowseNew(), but TBrowseDB() is easier since it automatically assigns some instance variables (nBottom, nLeft, nRight and nTop are described in Table 1) based on the four parameters passed to it. When using TBrowseNew(), you must assign values to these instance variables separately, after the object has been created. We also define four LOCAL variables for storing new TBColumn objects which shall be created in Step 2.

Once a TBrowse object has been created, you can appropriately assign some of its exported instance variables.

Step 2: Create multiple TBColumn objects.

Next, we must instantiate the TBColumn class once for each **column** in the browse by calling the TBColumnNew() method function. The general form of calling TBColumnNew() is:

```
TBColumnNew () --> oTBColumnObject
```

where TBColumnNew() returns a new TBColumn object oTBColumnObject using the specified **column** heading text cHeading and data retrieval code block bBlock. The cHeading text appears directly above the **column**'s data when the browse is displayed. As for bBlock, notice the flexibility that using a code block for data retrieval purposes affords. Columns can be filled with any data generated by a code block. Remember that any sequence of expressions can be embedded in a code block, including a call to a user-defined function (UDF). This implies that the data which populates a particular **column** in a browse can be arbitrarily complex.

Once a TBColumn object is created, you can appropriately assign some of its exported instance variables.
Here's the code to perform STEP 2 for our example thus far:

```
oColumn1 := TBColumnNew("Last Name",  {||employee->last} )
oColumn2 := TBColumnNew("First Name", {||employee->first} )
oColumn3 := TBColumnNew("Hire Date",  {||employee->hire_date} )
oColumn4 := TBColumnNew("Salary",     {||employee->mo_sal} )
```

This code creates four new TBColumn objects. The data retrieval blocks tell TBColumn in each case to simply get the field's contents and place it in the **column**.

At this point, the TBColumn objects aren't yet connected to the TBrowse object. This will be done in Step 3. Note that, depending on how many TBColumn objects are needed, you may want to generalize the code by saving the objects in an array. If you recall from a prior **Clipper Basics column** (see my November **column**, Data Base Advisor), **Clipper** arrays can contain values of any type, including objects.

Step 3: Inform the TBrowse object.

We need to inform the TBrowse of the TBColumn objects just created. The general way to do this is:

```
oTBrowseObject:addColumn(oTBColumnObject)
```

For some actual code, consider the following:

```
oEmpBrowse:addColumn( oColumn1 )
oEmpBrowse:addColumn( oColumn2 )
oEmpBrowse:addColumn( oColumn3 )
oEmpBrowse:addColumn( oColumn4 )
```

Using OOP jargon, each of the above statements is read: "send the addColumn() message to the oEmpBrowse object". In doing so four times, each with a different TBColumn object, we can create a four-**column** browse. If you save the objects in an array, a FOR loop may make for more elegant code.

If you need all fields of the database included in the browse, the above code may be generalized by the following UDF:

```
/***
* BrowseAll (oBrowse) --> oBrowse
* Send addColumn() message to passed
* TBrowse object for each field of
* currently selected database */

FUNCTION BrowseAll (oBrowse)
   LOCAL nFld, oColumn
   FOR nFld := 1 TO FCOUNT()
      * Create TBColumn object for current field.
      * Column heading shall be field name and
      * retrieval block is {|| fieldname}
      oColumn := TBColumnNew( FIELDNAME(nFld),;
                              FIELDBLOCK(FIELDNAME (nFld),;
                              SELECT() ) )

      oBrowse:addColumn( oColumn )
   NEXT
RETURN ( oBrowse ) // Return newly "informed" object
```

BrowseAll() assumes that the database is open and selected and that the TBrowse object exists. The FOR loop goes through each field of the database and, using the FIELDNAME() built-in function, supplies the TBColumnNew() constructor with its necessary parameters. We use FIELDBLOCK() to build the required data retrieval block. Finally, the addColumn() message is sent to the browse object, passing to it multiple TBColumn objects as the loop progresses. Notice that the TBColumn objects are never saved, just sent to the TBrowse object.

One last note about Step 3: This is a good place to tell TBrowse characteristics about the browse. Namely, you can specify what separator characters to use, colors, etc. To do this, you may assign new values to the appropriate TBrowse exported instance variables.

Step 4: Establish the primary TBrowse loop

The last step, and probably the most crucial, requires you to build a loop structure to control the operation of the browse. This loop must perform several tasks.

1. First we must "stabilize" the browse. The process of stabilization involves sending the stabilize() message to a TBrowse object, which tells the object to display current information on the screen. When you first create a TBrowse object, nothing is displayed on the screen. You must invoke the stabilize() method function which displays the headers. If you send the stabilize() message again, another row of the browse appears. This process proceeds until everything is displayed. The stabilize() method returns a .T. value when there's more to display, otherwise it returns .F. The most basic form of stabilization is to enclose stabilize() in a tight loop:

```
WHILE !oEmpBrowse:stabilize ()
END
```

This loop simply updates the screen and when done, resumes inside the primary loop.

2. The second task is to get a keystroke from the user. We must allow the user to move through the data. We need to trap keystrokes such as UP ARROW, HOME, ESCAPE, PAGE DOWN, etc. This task is achieved with the INKEY(0) function call as in:

```
nKey := INKEY(0) // Get a keystroke
```

3. Next, we must tell the browse how to respond to specific user requests (keystrokes). Since we normally need to handle a wide variety of keystrokes, the usual way to do this is to construct one long CASE statement where each CASE identifies a keystroke (usually chosen from the INKEY.CH header file). The following example only allows the user to move up and down through records:

```
DO CASE
   CASE nKey == K_UP    // UP ARROW key
      oEmpBrowse:up()  // Send up () message
   CASE nKey == K_DOWN  // DOWN ARROW key
      oEmpBrowse:down() // Send down () message
ENDCASE
```

We trap both the UP ARROW and DOWN ARROW keys in the CASE. Then, changes to the browse shall be displayed during the next stabilization.

4. The last step in the loop is to repeat everything until an exit is requested by the user.

Pulling it all together

Now let's look at an example that implements all four steps. The following code segment shows a modular form of a primary TBrowse loop. Stabilization is done with a UDF call to GenStab(). This way, you can do any kind of stabilization. Next, we grab a keystroke and arbitrarily choose ESC to mean exit. Finally, we do a call to another UDF, GenKey(), which becomes a keystroke handler.

```
#define TRUE .T.
#define NUM_COLS 4
#include 'inkey.ch'

FUNCTION Main
LOCAL oEmpBrowse, aColObjects[NUM_COLS]
LOCAL nKey
LOCAL cScrSave, nCnt
USE employee NEW
* Use constructor function to
* create 1 TBrowse object
oEmpBrowse := TBrowseDB ( 5, 10, 20, 70 )
* Use constructor function to
* create 4 new TBColumn objects
aColObjects[1] := TBColumnNew ( "Last Name",  {|| employee->last} )
aColObjects[2] := TBColumnNew ( "First Name", {|| employee->first} )
aColObjects[3] := TBColumnNew ( "Hire Date",  {|| employee->hire_date} )
aColObjects[4] := TBColumnNew ( "Salary" ,    {|| employee->mo_sal} )

* Tell TBrowse object about the TBColumn objects
FOR nCnt := 1 TO LEN(aColObjects)
   oEmpBrowse:addColumn(aColObjects[nCnt])
NEXT

* Customize TBrowse object with separators
oEmpBrowse:headSep := CHR (196)
oEmpBrowse:colSep := CHR (179)

* Save screen contents where browse will appear
cScrSave := SAVESCREEN( 5, 10, 20, 70 )

* Begin primary TBrowse loop
WHILE TRUE
   GenStab( oEmpBrowse ) // Perform generic stabilization
   nKey := INKEY (0) // Get a keystroke
   IF nKey == K_ESC // ESC is a good // exit key
      EXIT
   ELSE
      * Perform generic key handling
      GenKey ( oEmpBrowse, nKey )
   ENDIF
END

RESTSCREEN ( 5, 10, 20, 70, cScrSave)
```

```
RETURN NIL

/*** * GenStab( oTBObject ) --> NIL
*/
FUNCTION GenStab( oTBObject )
   WHILE !oTBObject:stabilize()
   END
RETURN NIL

/*** * GenKey( oTBObject, nKey ) --> NIL
*/
FUNCTION GenKey( oTBObject, nKey )
   DO CASE
      CASE nKey == K_UP
         oTBObject:up()
      CASE nKey == K_DOWN
         oTBObject:down()
      CASE nKey == K_PGUP
         oTBObject:pageUp()
      CASE nKey == K_PGDN
         oTBObject:pageDown()
      CASE nKey == K_LEFT
         oTBObject:left ()
      CASE nKey == K_RIGHT
         oTBObject:right ()
   ENDCASE
RETURN NIL
```

The browse screen that results from the above code is shown in Fig. 1.

Custom browses

With all this work, all we've achieved is a simple browse. This might seem like overkill to xBase or even Summer '87 programmers used to a BROWSE command or DBEDIT(). The advantage to this approach: With the object orientation surrounding database browsing in **Clipper**, we have the framework to do much more than other xBase languages. The example should give you a feeling for how customizable browsing with TBrowse and TBColumn works.

Let's cover some housekeeping chores. First, unless you specify otherwise, TBrowse builds a "vanilla" browse--only data values appear underneath the **column** headings you specified in the calls to TBColumnNew(). There are no boxes, borders, or colors. Moreover, the prior screen contents aren't saved. These are tasks you must perform.

Set conclusion on

We've just explored the tip of a very deep TBrowse iceberg. In a future **column**, we'll wrap up our basic introduction to TBrowse by looking a little deeper to find out how we can get much more out of **Clipper**'s browsing technology.

http://www.accessmylibrary.com/coms2/summary_0286-9273393_ITM