

An In-Depth Exploration of CA-Visual Objects New DataListView Class

Ed Richard

*CA-World 2000
eBusiness Solutions in Internet Time
JP185SN*

Introduction

In this session and article I will try to explain the full scope of this new class, introduced in CA-Visual Objects version 2.5. I will do this using a tutorial method; we will go through the various options and possibilities step by step. Each step will result in a working sample. All of the samples, and several others I made or found regarding the DataListView class, are included in the sample code for this session, as are the data files that are involved.

When you read the help file you will find that CA advertises this class as being designed as a lightweight read-only alternative to the DataBrowser. It is not able to handle direct (cell) editing, but notification is supported. This way, edits made in other places to the same DataServer are automatically visible. When you want to find out more about this notification, please take the time to read about it in the *Programmer's Guide*. I also explained it in my session, 'Making The Most of the Class Libraries,' presented at CA-World last year (1999).

To show you how easy and fast you can get up and running with the DataListView, we will start by building a browser form with an attached edit form.

Step 1: A Simple DataListView and an Attached Edit Form

First of all, we need a framework and a DataServer. We'll take a basic DataWindow application from the Basic tab in the Application Gallery and change the name to DataLv1. Add a module that will hold the generated DBServer code (Servers). Generate two DBServer subclasses here: one for Todo.Dbf and a second one for Person.Dbf. We will use both DataServers during this session. Load the main window in the Window module in the Window Editor. Place the necessary controls on the window using the Auto Layout feature and put an extra DataListView control on the form, changing its name to TodoDlv. When we have done this, all we need to do is tell the DataListView to use the same DataServer object the window uses. The PostInit method is the most appropriate place to do this:

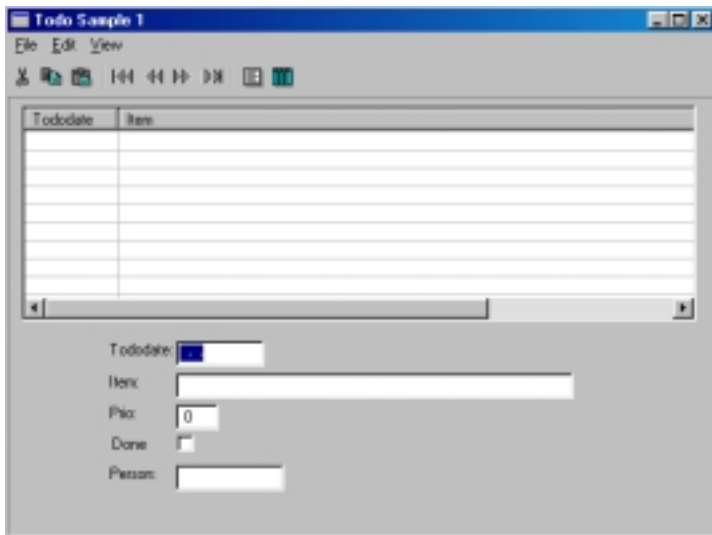
```
METHOD PostInit(oWindow,iCtlID,oServer,uExtra) CLASS MainWindow
//Put your PostInit additions here
SELF:oDcTododlv:Use( SELF:Server )
RETURN NIL
```

We can add this code by hand using the Source Code Editor or by clicking on the PostInit Actions property in the Window Editor.

We can now save the window and see if it works, but we have to make sure the application uses the proper defaults for RDD and Path. So add the following two lines to our Start method:

```
RDDSETDEFAULT( "DBFCDX" )
SetDefault( "C:\Cavo25\Samples\TsSamples\Datalistview\ " )
```

When we run this application, it should look like this:



Use Edit, Insert to add a record and you will see the result. As soon as you enter a value and tab off the SingleLineEdit control, the DataListView also gets updated. This is done by the DBServer and DataListView classes' implementation of notification.

Step 2: Adjusted DataListView Columns

As you have seen in the previous step, the DataListView has an auto layout mechanism, just like the DataWindow and DataBrowser classes. Therefore, we don't have to specify any column information in the Window Editor. Obviously, most of the time we do not want this. In the sample, the column for Item should be narrower, and we would like to see proper captions for the columns. Furthermore, we would like the place the columns in a different order, for example, the priority column should come first. Auto layout only occurs when the DataListView object has no columns at the time the Use method gets called. So we can add our columns before that happens. Since it is most likely that we will need a ListView like this in several places in the application, inheritance is the way to go here. We will create a DataListView subclass and add columns configured to our liking in the Init method. Once we have a subclass, we can load our window in the Window Editor, select the DataListView and specify that the InheritFrom property use the subclass in place of the DataListView class.

Here's the code for the Tododatalistview class:

```
CLASS Tododatalistview INHERIT DATALISTVIEW

METHOD Init(oOwner, xID, oPoint, oDimension, kStyle) CLASS
Tododatalistview
LOCAL oCol AS ListviewColumn
// Pass parameters as required by datalistview
SUPER:Init(oOwner, xID, oPoint, oDimension, kStyle)

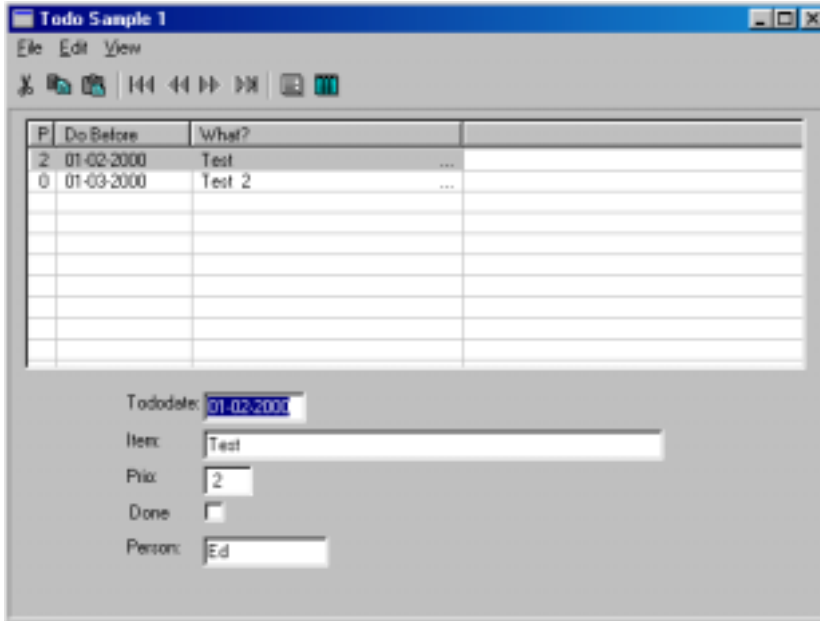
// First column, Priority, field as called Prio in Todo.Dbf
oCol := Listviewcolumn{2,#Prio}
oCol:Caption := 'P'
SELF:Addcolumn(oCol)

// Second column, Date, field as called Tododate in Todo.Dbf
oCol := Listviewcolumn{10,#Tododate}
oCol:Caption := 'Do Before'
SELF:Addcolumn(oCol)

// Third column, Text, field as called Item in Todo.Dbf
oCol := Listviewcolumn{20,#ITEM}
oCol:Caption := 'What?'
SELF:Addcolumn(oCol)

RETURN SELF
```

And here's what our form looks like now:



Step 3: The Usage of 'Virtual Columns'

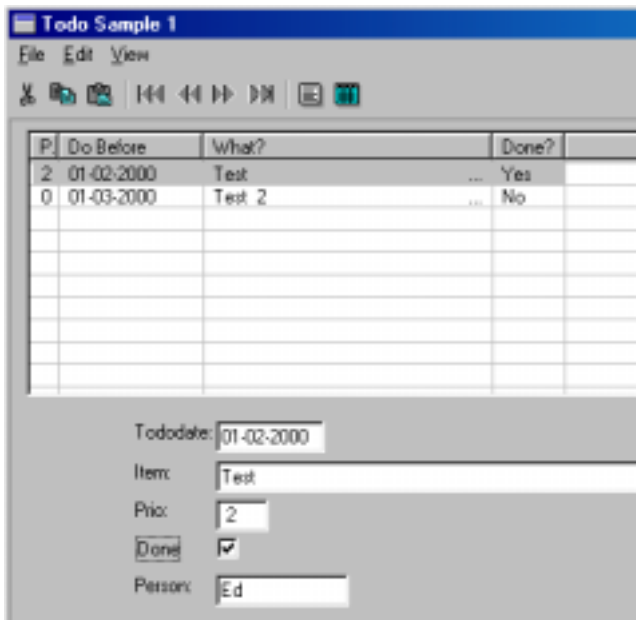
The DataListView class also comes with a mechanism to show information not contained in the fields of the attached DataServer directly. When we provide a FieldGet method in our subclass, we have the option to change anything we like with regards to cell content. It gets called for each cell that needs to be shown, and the name of the column is passed as a parameter. This way we can call `super:FieldGet(symName)` when we just want the field content, or do anything we like and return that value for other columns. Here again is some code, this time to show the words 'Yes' or 'No':

```
METHOD Init(oOwner, xID, oPoint, oDimension, kStyle) CLASS Tododatalistview
... Code as before
// Added a few lines here
// Fourth column, Done, Logic field as called Done in Todo.Dbf
oCol := Listviewcolumn{5,#DONE}
oCol:Caption := 'Done?'
SELF:Addcolumn(oCol)
```

```

RETURN SELF
METHOD FIELDGET(symColName) CLASS Tododatalistview
  IF symColName == #DONE
    IF SELF:server:FIELDGET(#Done)
      RETURN 'Yes'
    ELSE
      RETURN 'No'
    ENDIF
  ENDIF
  // All other columns are fine for now...
  RETURN SUPER:FIELDGET(symColName)

```



Again, as you can see, the column's content changes because of the notification even when all we do is click on the Done checkbox. This way we can achieve basically anything we like. I regularly use this to show information from a related DataServer.

Step 4: The Proper Order and Searching

In most ListViews in MS-Windows, the user can click a column header and the data is sorted. Usually this is not too difficult, because the ListView contains all of the data. In the case of the DataListView, the data you see is loaded but the rest is probably still in the .DBF, so sorting on-the-fly is not an option. In this case we can only use the available indices. When we have an index that corresponds with the column, it's really easy. All ListView events go through the owner window, so this is where these events can be handled. As we say in Holland, child's play (Tododlv Step 4a.Aef):

```
METHOD ListViewColumnClick(oListViewColumnClickEvent) CLASS MainWindow
LOCAL symColName AS SYMBOL
    SUPER:ListViewColumnClick(oListViewColumnClickEvent)

    symColName := oListViewColumnClickEvent:Listviewcolumn:Namesym
    SELF:server:Setorder(Symbol2String(symColName))
    RETURN NIL
```

Again, the notification takes care of the details and everything is handled automatically. A nice feature of the DBFCDX-driver is the ability to change the order of the indices on the fly from ascending to descending and vice versa. I just heard this is one of the new features in MS SQL Server 2000, go figure. To use this, we need some more code (Tododlv Step 4b.Aef):

```
METHOD ListViewColumnClick(oListViewColumnClickEvent) CLASS MainWindow
LOCAL symColName AS SYMBOL
    SUPER:ListViewColumnClick(oListViewColumnClickEvent)

    symColName := oListViewColumnClickEvent:Listviewcolumn:Namesym
    IF SELF:Server:Orderinfo(DBOI_NAME) # Symbol2String(symColName)
        SELF:Server:Setorder(Symbol2String(symColName))
    ELSE
        // Same order, let's reverse
        SELF:Server:OrderDescend(, !SELF:Server:OrderDescend())
        SELF:server:Notify(NOTIFYFILECHANGE)

ENDIF
RETURN NIL
```

Now while searching, it would be nice if the DataListView positions itself automatically as we type. Since we have the indices available this again is an easy task. Keyboard handling can be done at the control level, so called 'smart controls'. So implementing a key event handler at the DataListView level is our best option. The DataListView inherits from the ListView class, and this class has an internal keyboard buffer that it also uses for searches. Using this buffer will give us the standard Windows behaviour where it remembers the keystrokes until we pause, and then start over again. If you don't like this behaviour, we can always implement our own buffer and store the keystrokes at the Tododatalistview level.

Here is the code (Tododlv Step 4c.Aef):

```
METHOD KeyUp(oKeyEvent) CLASS Tododatalistview
  LOCAL liRecNo          AS LONGINT
  LOCAL xSeek AS STRING

  SUPER:KeyUp(oKeyEvent)
  IF oKeyEvent:ASCIICChar > 31 .and. oKeyEvent:ASCIICChar < 250
    liRecNo      := SELF:server:recno
    xSeek := SELF:SearchString
    IF !SELF:server:Seek(xSeek)
      SELF:Server:Goto( liRecNo )
    ENDIF
  //ELSEIF oKeyEvent:Keycode == KEYENTER
  //  SELF:MakeSelection()
  ENDIF
RETURN NIL
```

Some DataListViews have to show related data; you can easily apply a scope to the server. If that's the case, you will probably want to take that scope into account when searching. Here's a sample for that (Tododlv Step 4d.Aef):

```
METHOD KeyUp(oKeyEvent) CLASS Tododatalistview
  LOCAL liRecNo          AS LONGINT
  LOCAL xSeek AS STRING
  LOCAL xScope := SELF:Server:Ordertopscope AS USUAL

  SUPER:KeyUp(oKeyEvent)
  IF oKeyEvent:ASCIICChar > 31 .and. oKeyEvent:ASCIICChar < 250
    liRecNo      := SELF:server:recno
    IF !Empty(xScope)
      xSeek := xScope+SELF:SearchString
    ELSE
      xSeek := SELF:SearchString
    ENDIF
    IF !SELF:server:Seek(xSeek)
      SELF:Server:Goto( liRecNo )
    ENDIF
  ENDIF

RETURN NIL
```

Step 5: Using Images and Image Lists

Using images in a ListView requires some setting up before we can start. First of all, we need to have the images available, and we need icon subclasses for all images we will be using. We can do this by creating them using the Image Editor. (The module Images in TodoDlv Step 5.Aef contains them.) Next, we need to create an ImageList object that contains these icons. To be consistent with our object-oriented (OO) approach, we will subclass the ImageList class and create the list this way (again some code):

```
CLASS TodoImageList INHERIT ImageList
METHOD Init() CLASS TodoImageList
SUPER:Init(2,Dimension{16,16})
SELF:Add(v{ })
SELF:Add(d{ })

RETURN SELF
```

The ListView class has a SmallImageList property, and that's where we need to put our ImageList object:

```
METHOD Init(oOwner, xID, oPoint, oDimension, kStyle) CLASS
Tododatalistview
LOCAL oCol AS Listviewcolumn
SUPER:Init(oOwner, xId, oPoint, oDimension, kStyle)
SELF:smallimagelist := TodoImageList{ }
```

Now our DataListView is prepared for using the images. Next, we will add a column (in the Init method) to indicate if records are marked for deletion (don't forget to call SetDeleted(False)):

```
// Insert a column that shows deleted-status
oCol := ListViewColumn{2,#MARKEDDELETED}
oCol:Caption := ""
SELF:AddColumn( oCol )
```

The first column is used to show the images when a ListView is in report view mode, so the text or value doesn't matter, but we will use the value information to indicate which image to use. So we add some code to our FieldGet method to return a usable value (1 or 2, for example):

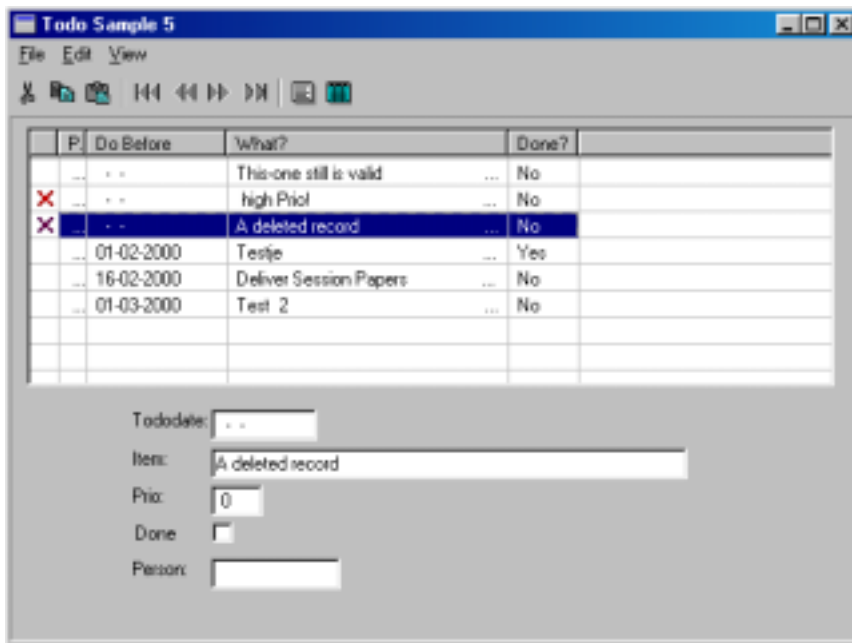
```
METHOD FIELDGET(symColName) CLASS Tododatalistview
DO CASE
CASE symColName = #MARKEDDELETED
RETURN IIF(SELF:Server:Deleted,1,0)
CASE symColName == #DONE
IF SELF:server:FIELDGET(#Done)
RETURN 'Yes'
ELSE
RETURN 'No'
ENDIF
ENDCASE

RETURN SUPER:FIELDGET(symColName)
```

Next we need to tell the ListView to use our images, depending on the status of the record we're on. After some studying of the SDK code, we found that `__GetDispInfo` does the trick:

```
METHOD __GetDispInfo(oCtrlNotifyEvent) CLASS Tododatalistview
    LOCAL di AS _winLV_DISPINFO
    LOCAL nRet AS LONG
    nRet := SUPER:__GetDispInfo(oCtrlNotifyEvent)
    di := PTR(_CAST, oCtrlNotifyEvent:lParam)
    IF _AND(di.Item.Mask, LVIF_IMAGE) == LVIF_IMAGE
        di.item.iImage := Val(di.item.PSZtext)
    ENDIF
    RETURN nRet
```

After implementing a Delete method at the DataWindow level, we can even toggle the deleted status using the Delete menu option.

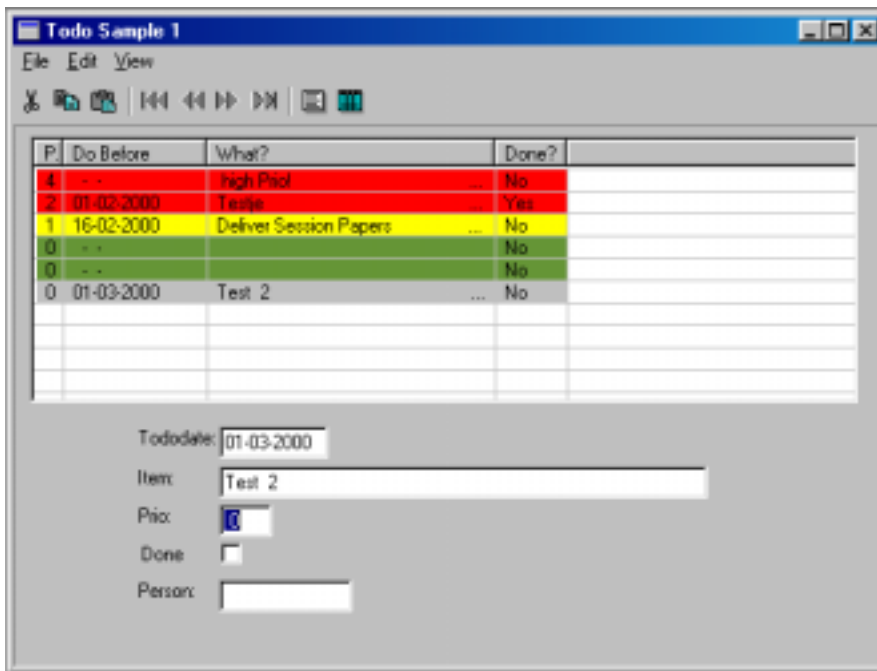


Step 6: Coloring Cells

Watch out when implementing this; I've noticed that on some computers, I don't know why yet, using both a `__GetDispInfo` and a `ControlNotify` message results in strange crashes.

Using colors is not something native to the `ListView` or `DataListView` class, but all controls send so-called `ControlNotify` messages to their owners. Using these events and reacting on some of them, you can certainly do things like changing colors. The `ListView` objects send 'Customdraw' events to their owner; this is when you can jump in and do your stuff. `TodoDlv Step 6.Aef` implements this. Notice we have to set the background of the `ListView` to yellow to color the area not covered by data.

`TodoDlv 6b` even implements the coloring of cells depending on values:



The Final Step: Some Tips and Tricks

Handling of Deleted Records

When `SetDeleted(TRUE)`, the `DataListView` initially doesn't show records marked for deletion. However, when you delete a record the `DataListView` is actually showing, it doesn't handle it properly. A workaround for this is implementing a filter that ignores deleted records:

```
SELF:server:Setfilter({|.not.DELETED()|}, ".not. Deleted()")
```

Conclusion

Well, there isn't much more I can think of for `DataListViews`, but I'm surely interested if you have any suggestions. It would be nice to have a document like this that covers everything there is to know about the `DataListView` class, so just let me know when you make some enhancements or have something you would like it to do but can't figure out how.

Bibliography

Included are several .AEF files for the above saved in different stages; I've also made sure that the .DBF files used are included.

CA-Visual Objects 2.5 Programmer's Guide. Computer Associates International.

CA-Visual Objects 2.5 SDK. Computer Associates International.

Special thanks go to Robert van der Hulst and Erik Visser for helping me with some of the fine-tuning of the samples while I was doing the R&D for this session.

Ed Richard is co-founder and co-owner of T&S ObjecTS, a Netherlands-based company that specializes in application development, consulting and training in CA-Visual Objects. Ed has been teaching students CA-Clipper and CA-Visual Objects since 1990 and has been running a team of programmers developing in these environments over the same period.

T&S provides Computer Associates in the Netherlands with a professional helpdesk for CA-Clipper and CA-Visual Objects.

Ed is a frequent speaker at SDGN meetings and at the yearly 'Conference To The Max', organized by SDGN. Ed is also President of the Dutch CA-Visual Objects section of SDGN (Software Developers Group Netherlands). The Dutch User Group has a very active CA-Visual Objects section, with a great magazine and a yearly conference. You can check out SDGN at www.sdgn.nl. Ed can be reached at erichard@tobjects.nl.