Browsing Your Data in an n-Tier Application

Allan Wissing

CA-World 2000
eBusiness Solutions in Internet Time
JI045SN

Introduction

Data browsers are a well-accepted user interface in applications written today. But what happens when the data is several hundred miles away and we want to provide the user with a picklist of items from the database?

This session addresses the design concept of our data browser. We will determine its basic structure, provide the ability to gather data from afar, and scroll through that data. We will then add the ability to search for a specific record that might possibly involve specifying a different index order. Finally, we will show how to reuse our browser so that it can maintain the data it is attached to, or be a popup window for a SingleLineEdit, so that the user can select the desired record.

So Let's Get Started

We want to develop a 3-tier application, which in this case will require two EXE files. The first is my Human Resource (HR) Client application. This application resides on the desktop of our HR administrators (around the world), and they can query and edit the database.

The second application is the Data Server application. This application resides on an NT server machine located in New Zealand, and it is connected to the databases, which can be of any type, including Jasmine *ii* ODBMS and/or dBase.

The Client application requirements include the ability to search for an employee using either the employee number or the employee name. The user interface must be user friendly, as there are 10,000 employees to handle.

The Client application is connected to the Server application using DCOM technology.

The Array Server

The user interface we intend to use is a data browser. Normally a data browser is connected to a database, but that is not possible in this case because the table is too far away (in New Zealand).

The next best thing is to introduce an ArrayServer and use this as a medium to store our employee records. What we will do is populate the array server with a handful of records at a time.

The array server will have the same business logic as a data server. In fact, the ArrayServer inherits from the DataServer. However, instead of serving up records from a file, it serves up records that are stored in an array. This is achieved by giving our ArrayServer properties where it can store the database structure, associated FieldSpecs, and finally the data itself.

The ArrayServer class, which I am using as a base class, is not standard with CA-Visual Objects. However, it is available in the VOForum Library. This can be downloaded from http://www.knowvo.com. This class will also be made available, along with all other source code.

After checking out the ArrayServer class, we will proceed to build upon it (using inheritance). I will create an EmployeeArrayServer, as follows:

```
CLASS EmployeeArrayServer INHERIT ArrayServer
      HIDDEN oHRServer AS OBJECT      // This object is of Class iDispatch

METHOD Init(oHRServer) CLASS EmployeeArrayServer
   LOCAL aStructure, aServerContent AS ARRAY

   SELF:oHRServer := oHRServer
   SELF:SetOrder(1)
   oHRServer:SetUpEmployeeArrayServer("", ;
                                      SELF:IndexOrd(), ;
                                      SELF:BrowserArraySize, ;
                                      @aStructure, ;
                                      @aServerContent)

   SUPER:Init(aStructure)
   SELF:FillUsing(aServerContent)
RETURN SELF
```

On initializing our EmployeeArrayServer class, we establish the table structure and also the content. Note that we pass two variables by reference. The method call is to a method from our Server application.

```
METHOD SetUpEmployeeArrayServer(sSearchValue, ;
                                dwIndexOrd, ;
                                dwDBArraySize, ;
                                aStructure, ;
                                   aDBArray) CLASS ActiveXServerApp

   aStructure := SELF:oEmployeeDataServer:StructureRequiredForBrowser
RETURN SELF:GetDBArray(sSearchValue, dwIndexOrd, dwDBArraySize, @aDBArray)

ACCESS StructureRequiredForBrowser CLASS EmployeeDataServer
   // Returns a structure array, which is required by the DataBrowser
   LOCAL aStructure, aReturnVal AS ARRAY
   LOCAL dwEmployNum, dwEmployName AS DWORD

   aStructure := SELF:DBStruct
   dwEmployNum := SELF:FieldPos(#EmployNum)
   dwEmployName := SELF:FieldPos(#EmployName)

   aReturnVal := {aStructure[dwEmployNum], aStructure[dwEmployName]}
RETURN aReturnVal
```

When creating the structure of our EmployeeArrayServer, we want to give thought to which fields are required. What I want to present to the user is a DataBrowser that shows the Employee Number and the Employee Name. So the ArrayServer requires only these two fields. The routine above creates an array of the Employee database, and then it picks out the two fields it requires and only returns those fields to the client. So our structure is going to look something like this:

Name	Type	Length	Dec
EmployNum	C	8	0
EmployName	C	25	0

Then we send the records that are going to be stored in our EmployeeArrayServer back to the client.

```
METHOD GetDBArray(sSearchValue, ;
                  dwIndexOrd, ;
                  dwDBArraySize, ;
                  aDBArray) CLASS EmployeeDataServer
   LOCAL lFoundFirstTime AS LOGIC

   lFoundFirstTime := SELF:FindFirstRecord(sSearchValue, dwIndexOrd)
   aDBArray := SELF:BIdBrowserArray(dwDBArraySize)
RETURN lFoundFirstTime

METHOD FindFirstRecord(sSearchValue, dwIndexOrd) CLASS EmployeeDataServer
   LOCAL lFoundFirstTime AS LOGIC
   LOCAL dwLen AS DWORD

   SELF:SetOrder(dwIndexOrd)
   lFoundFirstTime := SELF:Seek(sSearchValue)

   IF !lFoundFirstTime
      // The SearchValue does not exist, so we will now trim the SearchValue
      // and keep trimming it until we have found what we are after
      dwLen := SLen(sSearchValue)
      // The Index Order has already been set, so just seek
      DO WHILE dwLen > 0 .AND. !SELF:Seek(sSearchValue)
         dwLen--
         sSearchValue := Left(sSearchValue, dwLen)
      ENDDO
   ENDIF

   IF Empty(sSearchValue)
      SELF:GoTop()
   ENDIF
RETURN lFoundFirstTime
```

```
METHOD BldBrowserArray(dwArraySize) CLASS EmployeeDataServer
   LOCAL dwCount AS DWORD
   LOCAL aReturnVal := {} AS ARRAY

   ASize(aReturnVal, dwArraySize)
   dwCount := 1
   DO WHILE !SELF:EOF .AND. dwCount <= dwArraySize
      aReturnVal[dwCount++] := {SELF:EmployeeNum, SELF:EmployeeName}
      SELF:Skip(1)
   ENDDO

   IF dwCount <= dwArraySize
      ASize(aReturnVal, dwCount - 1)
   ENDIF
RETURN aReturnVal
```

The first thing to note here is the **EmployeeDataServer:FindFirstRecord()** method, where we pass to it the seek value and the index order. This method then sets the index, does a seek, and stores the result to **IFoundFirstTime**. If the seek was not successful, the method starts trimming off the last character of the seek value and tries again until it is successful in finding a record. The record pointer is left in that position and we return the success of our initial seek.

The next method builds an array that contains an Employee Number and Employee Name. The first record is the record we left the record pointer at before, and we only allow the array to be of **dwArraySize** length. This parameter, you will note, was sent by the Client application, and in this case was 25.

So to sum it up, we have created an EmployeeArrayServer, which is storing the Employee Number and the Employee Name for 25 of our employees.

We can now attach this to our DataBrowser, which we are about to design.

Creating a Data Browser

DataColumns

A DataBrowser is basically a DataWindow with **SELF:ViewAs()** set to **#BrowseView.**

I will design the columns first:

```
CLASS EmployNameColumn INHERIT DataColumn

METHOD Init() CLASS EmployNameColumn
    SUPER:Init(19)
    SELF:HyperLabel := HyperLabel{#EmployName,"Name",NULL_STRING,NULL_STRING}
    SELF:Caption := "Name"

CLASS EmployNumColumn INHERIT DataColumn

METHOD Init() CLASS EmployNumColumn
    SUPER:Init(8)
    SELF:HyperLabel := HyperLabel{#EmployNum,"Num",NULL_STRING,NULL_STRING}
    SELF:Caption := "Num"
```

The HyperLabel is an important part here. The first parameter tells the browser which field from our data server to populate the column with. These names need to be the same as the field names on our EmployeeArrayServer.

The Captions will be used as the column headings on our DataBrowser.

DataBrowser

To create the EmployeeDataBrowser class, we do the following:

```
CLASS EmployeeDataBrowser INHERIT DataBrowser

METHOD Init(oOwner) CLASS EmployeeDataBrowser
   SUPER:Init(oOwner)
SELF:sSearchString := ""
SELF:AddColumn( EmployNumColumn{} )
   SELF:AddColumn( EmployNameColumn{} )
   SELF:SetStandardStyle(GBSREADONLY)
   SELF:ParentWindow := SELF:Owner
```

DataBrowser Container

I now require an object to store the EmployeeDataBrowser (a container, so to speak). This object will be a DataWindow object that will display the browser. Using the Window Editor, all I am really doing is creating an empty window with the name EmployeeDataBrowserContainer.

```
CLASS EmployeeDataBrowserContainer INHERIT DataWindow

METHOD Init(oWindow, iCtlID, oServer, uExtra) CLASS EmployeeDataBrowserContainer
   oServer := GetEmployeeArrayServer()
   SUPER:Init(oWindow,iCtlID,oServer,uExtra)
   SELF:Browser := EmployeeDataBrowser{ SELF }
   SELF:Browser:Use( oServer )
   SELF:ViewAs( #BrowseView )
```

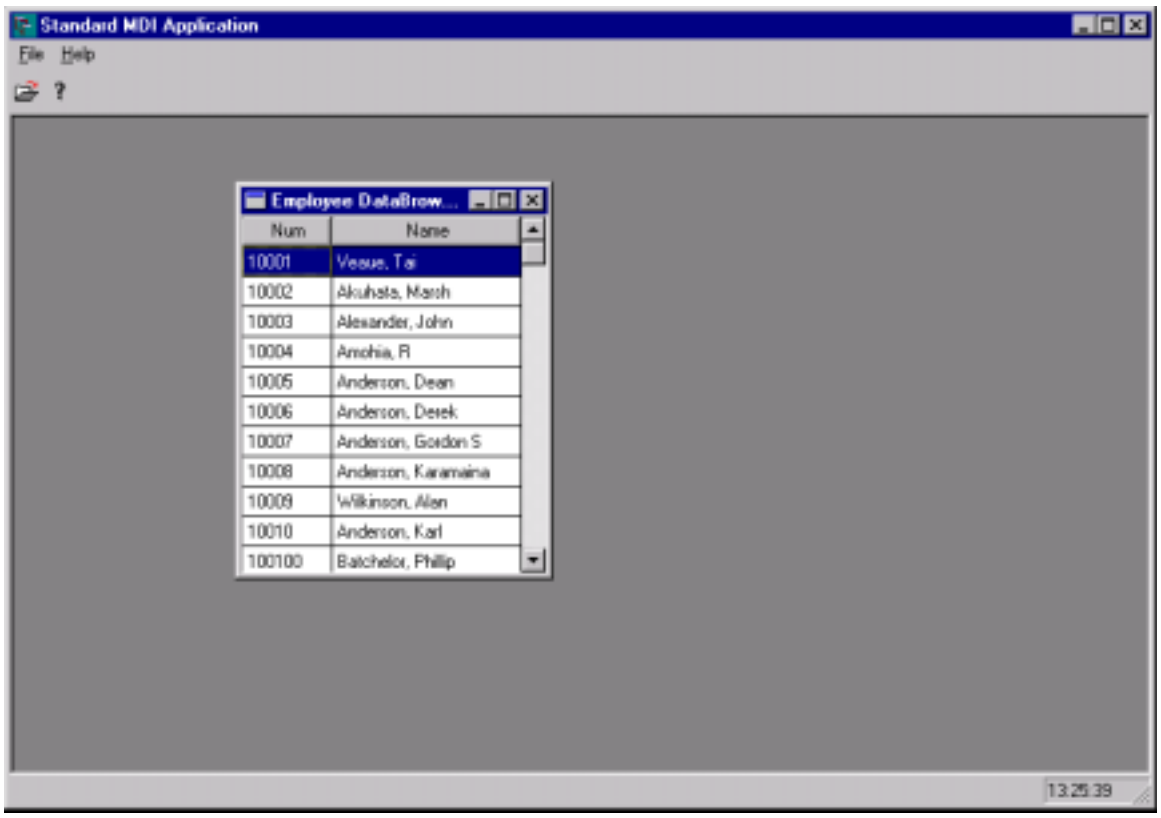
There are a few things that happen in the code above, which I must enlighten you on.

(a) After the SUPER:Init() I want to declare the EmployeeDataBrowser , it is important that we remain in #Formview (the default) while we do this.

(b) Immediately after the declaration we set up the Dataserver the browser is going to use, which in this case is our EmployeeArrayServer.

(c) Lastly, we change to Browse View.

And now in the first instance we have designed our DataBrowser. We can verify that the DataBrowser works by creating a menu item that calls our browser. When we open the window it will look like this:



We can now use this browser to scroll the contents of our EmployeeArrayServer. If we were to press the Down Arrow key, after 25 hits we would reach the bottom and the browser would go no further. This is because we only loaded our server with 25 records. We need the system to detect that we have reached the bottom and then fill the server with another 25 records from the database.

To achieve this, we can write our own browser Dispatch method.

```
METHOD Dispatch(oEvent) CLASS EmployeeDataBrowser
   LOCAL nReturn := 1L AS LONGINT
   LOCAL nKeyCode AS DWORD
   LOCAL iOldRecNum AS INT
   LOCAL oDataServer AS OBJECT

   IF oEvent:Message == WM_COMMAND
      nKeyCode := HiWord(oEvent:wParam)

      DO CASE
      CASE SELF:KeyCodeMovesToNewRecord(nKeyCode)
         SELF:ClearSearchString()

            oDataServer := SELF:oDataServer
            iOldRecNum := oDataServer:RecNo      // Record the Existing Record Number
            nReturn := SUPER:Dispatch(oEvent)    // Call SUPER:Dispatch()
                                                 //
         IF nKeyCode ==    CN_LK_ARROW_DOWN .and. ;
            iOldRecNum == SELF:nOldRecordNum
            // We have hit the last record
               SELF:FillBrowser(SELF:GetIndexOrderColumn:Value)
               IF oDataServer:RecNo < oDataServer:RecCount
                  oDataServer:Skip(1)
               ENDIF
         ENDIF

         OTHERWISE
      nReturn := SUPER:Dispatch(oEvent)
      ENDCASE
   ENDIF
RETURN nReturn
```

When we push the Down Arrow key, the Dispatch is called. Just prior to calling SUPER:Dispatch(), I record the current record number. When we return from SUPER:Dispatch(), our browser should reside on the next record, so we check the current record number against the old record number. If the numbers agree, then we have reached the bottom and we need to go away and retrieve another group of records.

```
METHOD FillBrowser(sSearchValue) CLASS EmployeeDataBrowser
    LOCAL oOldPointer AS Pointer
    LOCAL lFoundFirstTime AS LOGIC

    oOldPointer := SELF:oParentWindow:Pointer
    SELF:oParentWindow:Pointer := Pointer{POINTERHOURGLASS}
    lFoundFirstTime := Send(SELF:oDataServer, #FillDBArray, sSearchValue)
    SELF:Refresh()
    SELF:SetColumnFocus(SELF:GetIndexOrderColumn)
    SELF:oParentWindow:Pointer := oOldPointer
RETURN lFoundFirstTime
```

The FillBrowser method calls the array server to go get another 25 records and then does a SELF:Refresh(), which will force the browser to populate with the new contents of the array server.

Change the Index Order

The Employee database has two index orders, Employee Number and Employee Name. It would be nice to have the ability to view our data in either order. To achieve this, I am going to use the DataBrowser Focus feature. If you look at our DataBrowser as it stands, the selected row is highlighted. But if you look even harder, you will see that one of the cells in the selected row is surrounded by dotted lines. This is the column that has focus. If we were to press the Left Arrow or Right Arrow key, then the column focus would change. Using the ColumnFocusChange method, we are going to provide a user-definable index order.

```
METHOD ColumnFocusChange(oColumn, lHasFocus) CLASS EmployeeDataBrowser
    LOCAL nNewIndexOrder AS DWORD
    LOCAL lDisplayChange AS LOGIC

    lDisplayChange := TRUE
```

```
IF oColumn:NameSym == #EmployNum .AND. lHasFocus
    nNewIndexOrder := EmployeeNumberIndexOrder
ELSEIF oColumn:NameSym == #EmployName .AND. lHasFocus
    nNewIndexOrder := EmployeeNameIndexOrder
ELSE
    lDisplayChange := FALSE
    SUPER:ColumnFocusChange(oColumn, lHasFocus)
ENDIF

IF lDisplayChange .AND. !(Send(SELF:oDataServer, #INDEXORD) == nNewIndexOrder)
    SELF:SetOrder(nNewIndexOrder)
ENDIF
```

Now each time we change the column focus, we will also change the index order, and the EmployeeArrayServer will refresh its contents.

Searching the DataBrowser

Rather than scroll the data browser, it would be much nicer to have the ability to do a search for a specific customer. In fact, scrolling is to be frowned on because it may result in too many unnecessary calls to the server in New Zealand.

We will start by capturing the keystrokes while focus is on the data browser. This is achieved by enhancing the Dispatch method.

```
METHOD Dispatch(oEvent) CLASS EmployeeDataBrowser
    LOCAL nReturn := 1L AS LONGINT
    LOCAL nKeyCode AS DWORD
    LOCAL iOldRecNum AS INT
    LOCAL oDataServer AS DataServer

    IF oEvent:Message == WM_COMMAND
        nKeyCode := HiWord(oEvent:wParam)
        DO CASE
            CASE nKeyCode == CN_CHARHIT
                SELF:SearchString += Chr(CntCNCharGet(SELF:Handle(), oEvent:lParam))
                SetTimer(SELF:Handle(), TimeToSeekOutSearchString, 1000, NULL_PTR)
            …
```

When we identify there has been an interesting character hit, we then add it to our search string. At the same time, we set the timer for when we want to do the search. I have set the timer for 1 second. The purpose for doing this, is that we do not want to rush off to the server every time there is a keystroke. Instead, we wait a second after the last keystroke before we go to the server to do our search.

The Dispatch is again responsible for activating the search, as follows:

```
...
    ELSEIF oEvent:Message == WM_TIMER
        DO CASE
        CASE oEvent:wParam == TimeToSeekOutSearchString
                KillTimer(SELF:Handle(), TimeToSeekOutSearchString)
            SELF:FillBrowserWithSearchString()
...

METHOD FillBrowserWithSearchString() CLASS EmployeeDataBrowser
    SELF:FillBrowser(SELF:SearchString)
    SELF:EstablishCurrentSearchValue()
    SetTimer(SELF:Handle(), TimeToEraseSearchString, 5000, NULL_PTR)
```

The Dispatch method calls the FillBrowserWithSearchString method, which calls our FillBrowser method, passing the SearchString as a parameter. After having repopulated the browser with the new array, we need to test whether our search string is the same as the closest match we made when searching the database because we want to display the result. Lastly, we want a timer set to remove the search string if no buttons have been pushed for 5 seconds.

First, let's work on the bit of code that displays the result of our search string:

```
ASSIGN SearchString(cString) CLASS EmployeeDataBrowser
    sSearchString := cString
    KillTimer(SELF:Handle(), TimeToEraseSearchString)
    SELF:ShowSearchText()
```

Each time we push an interesting key, the Dispatch method assigns a value to SearchString, which means this assign is called. The assign not only stores the value, but also stops the timer that will erase the string and calls the ShowSearchText method.

```
METHOD ShowSearchText() CLASS EmployeeDataBrowser
    LOCAL sMsg AS STRING
    LOCAL oStatusBar AS StatusBar

    sMsg := ""
    oStatusBar := oParentWindow:StatusBar
    IF oStatusBar == NULL_OBJECT
        oStatusBar := GetShellWindow():StatusBar
    ENDIF

    IF oStatusBar <> NULL_OBJECT
        IF !Empty(SELF:SearchString)
            sMsg := "Searching for: " + SELF:SearchString
        ENDIF
        oStatusBar:PermanentText := sMsg
    ENDIF
```

If oParentWindow has a status bar, I want to print the result of my search on that status bar. If there is no status bar on oParentWindow, I will instead print the result on the ShellWindow status bar. oParentWindow is declared in the EmployeeDataBrowser:Init() method, and it also has an assign so that the owner window can modify its value (See below).

So now we have the ability to do a keyboard search, which means we can key in an Employee Name or Employee Number and the system will locate the desired records and redisplay them on the browser.

Inheritance

Our DataBrowser, as it currently exists, is a standalone object that can think for itself. It knows what to do when it has to initialize itself, and when the browser has focus and the user presses keys, it knows how to react to those keyboard instructions.

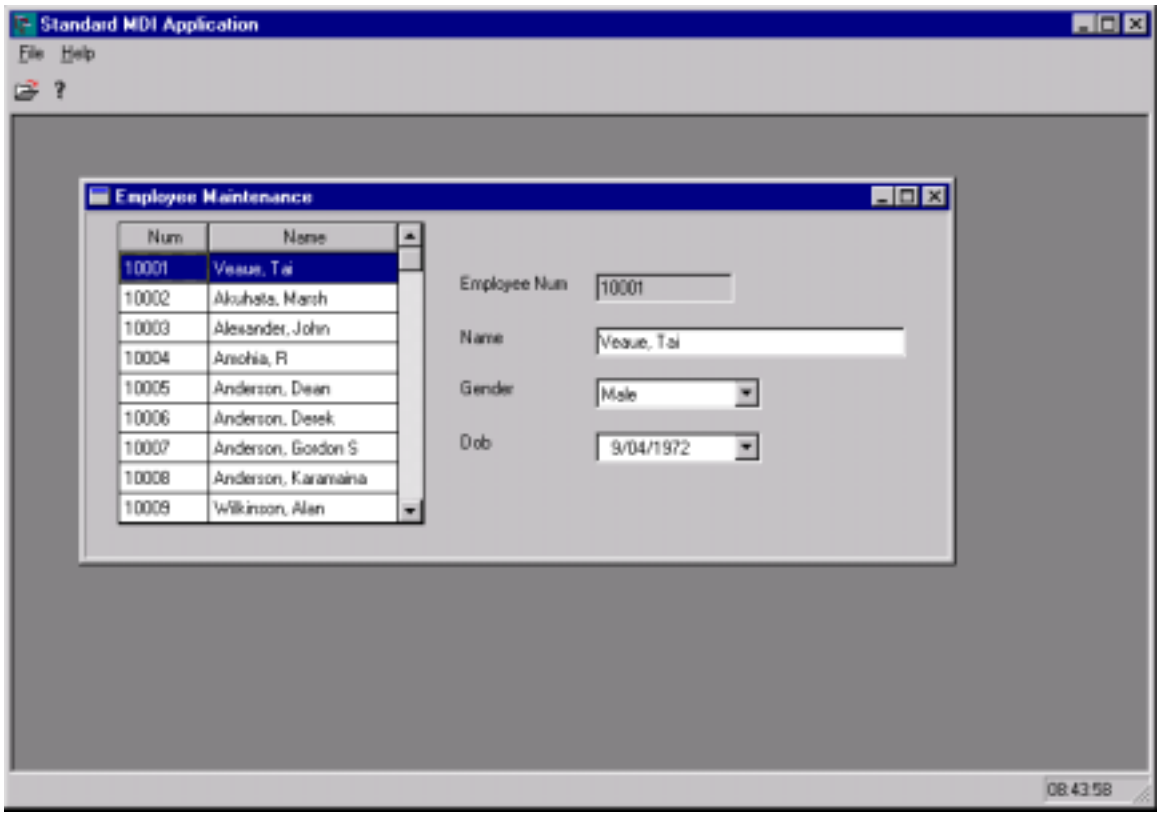
Now that we have designed our DataBrowser, it's time to put it to good use. We now have a data window that we can call on whenever we wish to find an employee.

The first example is to design an Employee Maintenance Window, where we can call up an employee, change his details, then save the result.

The second example is to search for an employee whose name is required to populate a SingleLineEdit control.

Employee Maintenance DataWindow

The data window I am going to design will look like the one illustrated below. The data browser on the left is a SubDataWindow on this new window. The name of the SubDataWindow is going to be EmployeeDataBrowserContainer, which is the name of our data window we have designed. I have populated the right hand side of this data window with controls to enable editing of employee details.



If we were to test our application now, we would discover our browser was operating as expected. However, we would like to have the controls on the right hand side updated each time the browser rests on a new record.

Firstly, I want to be able to identify which window we want the browser to communicate with, so I need an access in the EmployeeDataBrowser:

```
ASSIGN ParentWindow( o ) CLASS EmployeeDataBrowser
    // o  - Is the Window, that we want the browser to communicate with
    SELF:oParentWindow := o
    IF (SELF:INotifyParentWhenBrowserPointerRestsOnNewRecord := ;
            IsMethod(o, #BrowserPointerRestsOnNewRecord)
        Send(o, #BrowserPointerRestsOnNewRecord, SELF)
    ENDIF
```

Our new data window can now call this access and register itself as the ParentWindow, and thus it requires notification. At the same time, the access tests to see if the method BrowserPointerRestsOnNewRecord exists. If it does, then this method will be called when the browser rests on a record. The functionality that enables this behavior is the Notify method.

```
METHOD Notify(kNotifyName, uDescription) CLASS EmployeeDataBrowser
    LOCAL uValue AS USUAL
    LOCAL dwTimePeriod AS DWORD

    uValue := SUPER:Notify(kNotifyName, uDescription)

    //Put your changes here
    IF SELF:INotifyParentWhenBrowserPointerRestsOnNewRecord .AND. ;
        kNotifyName == NOTIFYRECORDCHANGE
        dwTimePeriod := 500
        SetTimer(SELF:Handle(), TimedBrowserResting, dwTimePeriod, NULL_PTR)
    ENDIF

    RETURN uValue
```

The Notify method detects that there has been a record change, but again, because there can be many record changes in a short space of time (e.g., if the operator holds down an arrow key), we get the timer to wait a ½ second before taking action.

```
METHOD Dispatch(oEvent) CLASS EmployeeDataBrowser
    …
        CASE oEvent:wParam == TimedBrowserResting
        KillTimer(SELF:Handle(), TimedBrowserResting)
            Send(oParentWindow, #BrowserPointerRestsOnNewRecord, SELF)
    …
```

The action that we take is to call the BrowserPointerRestsOnNewRecord method of the ParentWindow, which we previously registered. In this case, the method simply calls the Scatter method.

Retrieving Data From a Server Application

The process of retrieving data from the Server application should be as painless as possible. Retrieving data from a server and saving data back to the server is a repetitive process. Each time we retrieve data, we need to collect the data, sort it, wrap it, and send it, and on its arrival at the client side, it needs to be unwrapped and the data dispersed. One of the best methods of achieving this is by using object orientation techniques.

In the source code that accompanies this lecture you will note I have a Common HR Library. This library contains classes and functions that the entire application needs access to. One such class is the EmployeeDetails class:

```
CLASS EmployeeDetails
   HIDDEN aContent[Emp_Det_LastItem] AS ARRAY

   DECLARE ACCESS Content
   DECLARE ASSIGN Content
   DECLARE ACCESS DateofBirth
   ....

ACCESS Content AS ARRAY PASCAL CLASS EmployeeDetails
RETURN SELF:aContent

ASSIGN Content(a AS ARRAY) AS ARRAY PASCAL CLASS EmployeeDetails
RETURN SELF:aContent := a

ACCESS DateofBirth AS STRING PASCAL CLASS EmployeeDetails
RETURN SELF:aContent[Emp_Det_DOB]

ACCESS EmployeeNum AS STRING PASCAL CLASS EmployeeDetails
RETURN SELF:aContent[Emp_Det_EmployNum]

ACCESS Gender AS STRING PASCAL CLASS EmployeeDetails
RETURN SELF:aContent[Emp_Det_Gender]

METHOD ExtractContent(o AS OBJECT) PASCAL CLASS EmployeeDetails
   o:FieldPut(#EmployNum, SELF:aContent[Emp_Det_EmployNum])
   o:FieldPut(#EmployName, SELF:aContent[Emp_Det_Name])
   o:FieldPut(#Gender, SELF:aContent[Emp_Det_Gender])
   o:FieldPut(#BirthDate, SELF:aContent[ Emp_Det_DOB])
```

```
METHOD FillContent(o AS OBJECT) PASCAL CLASS EmployeeDetails
    SELF:aContent[Emp_Det_EmployNum] := Trim(o:FieldGet(#EmployNum))
    SELF:aContent[Emp_Det_Name] := Trim(o:FieldGet(#EmployName))
    SELF:aContent[Emp_Det_Gender] := o:FieldGet(#Gender)
    SELF:aContent[Emp_Det_DOB] := o:FieldGet(#BirthDate)

STATIC DEFINE Emp_Det_EmployNum := 1
STATIC DEFINE Emp_Det_Name := 2
STATIC DEFINE Emp_Det_Gender := 3
STATIC DEFINE Emp_Det_DOB := 4
STATIC DEFINE Emp_Det_LastItem := 4
```

What we are looking at here is a container to package and transport the data from the Server to the Client and vise versa. But what's that I hear you say? "Passing objects across the wire is not efficient, because …" - Yes, but check out the code on the Server application and then look at the Client code.

The Server does the following:

```
METHOD GetEmployeeDWContent(sEmployNum) CLASS ActiveXServerApp
RETURN SELF:oEmployeeDataServer:GetEmployeeDWContent(sEmployNum)

METHOD GetEmployeeDWContent(sEmployNum) CLASS EmployeeDataServer
    LOCAL oEmployeeDetails AS EmployeeDetails

    SELF:SetOrder(1)
    SELF:Seek(sEmployNum)
    oEmployeeDetails := EmployeeDetails{}
    oEmployeeDetails:FillContent(SELF)
RETURN oEmployeeDetails:Content
```

Note that the Server returns oEmployeeDetails:Content. If we look at the code above and check out the access for Content, you will note that it returns an array. It is this array that is passed back to the Client application.

Now let's check out the Client application.

```
METHOD Scatter() CLASS EmployeeDW
    LOCAL aContent AS ARRAY
    LOCAL oEmployeeDetails AS EmployeeDetails
    LOCAL sEmployNum AS STRING
```

```
sEmployNum := oSFEmployeeDataBrowserContainer:Server:FieldGet(#EmployNum)
aContent := GetHRServer():GetEmployeeDWContent(sEmployNum)
oEmployeeDetails := EmployeeDetails{}
oEmployeeDetails:Content := aContent
oEmployeeDetails:ExtractContent(SELF)
...
```

The Client application retrieves the array from the Server application, initializes the EmployeeDetails class, then stores the array using EmployeeDetails:Content.

The EmployeeDetails class is the same class we used to create the array – it is the class sitting in our common library. We now have a container where we can query the values and assign them to a data window control.

The final call in the method above is to ExtractContent. It reads like this:

```
METHOD ExtractContent(o AS OBJECT) PASCAL CLASS EmployeeDetails
    o:FieldPut(#EmployNum, SELF:aContent[Emp_Det_EmployNum])
    o:FieldPut(#EmployName, SELF:aContent[Emp_Det_Name])
    o:FieldPut(#Gender, SELF:aContent[Emp_Det_Gender])
    o:FieldPut(#BirthDate, SELF:aContent[Emp_Det_DOB])
```

In the Scatter method we pass SELF (our DataWindow) to the method, and the controls are then filled with the values from the array. What may not be so obvious, however, is that the symbol names are the same as the database field names.

This method is therefore reusable later on when we want to save the contents of the data window in the Server application.

This same feature is also present in the FillContent method, and this is again because the control names in my data window are the same field names as used in the Employee database.

```
METHOD FillContent(o AS OBJECT) PASCAL CLASS EmployeeDetails
    SELF:aContent[Emp_Det_EmployNum] := Trim(o:FieldGet(#EmployNum))
    SELF:aContent[Emp_Det_Name] := Trim(o:FieldGet(#EmployName))
    SELF:aContent[Emp_Det_Gender] := o:FieldGet(#Gender)
    SELF:aContent[Emp_Det_DOB] := o:FieldGet(#BirthDate)
```

Explore this further when you get the source code.

Editing our Employee Details

What we have now is a data window, where controls are populated from the database values associated with the current record in the data browser. We may want to edit these values, and as you can see, it is easy to change focus to a control and change the values therein.

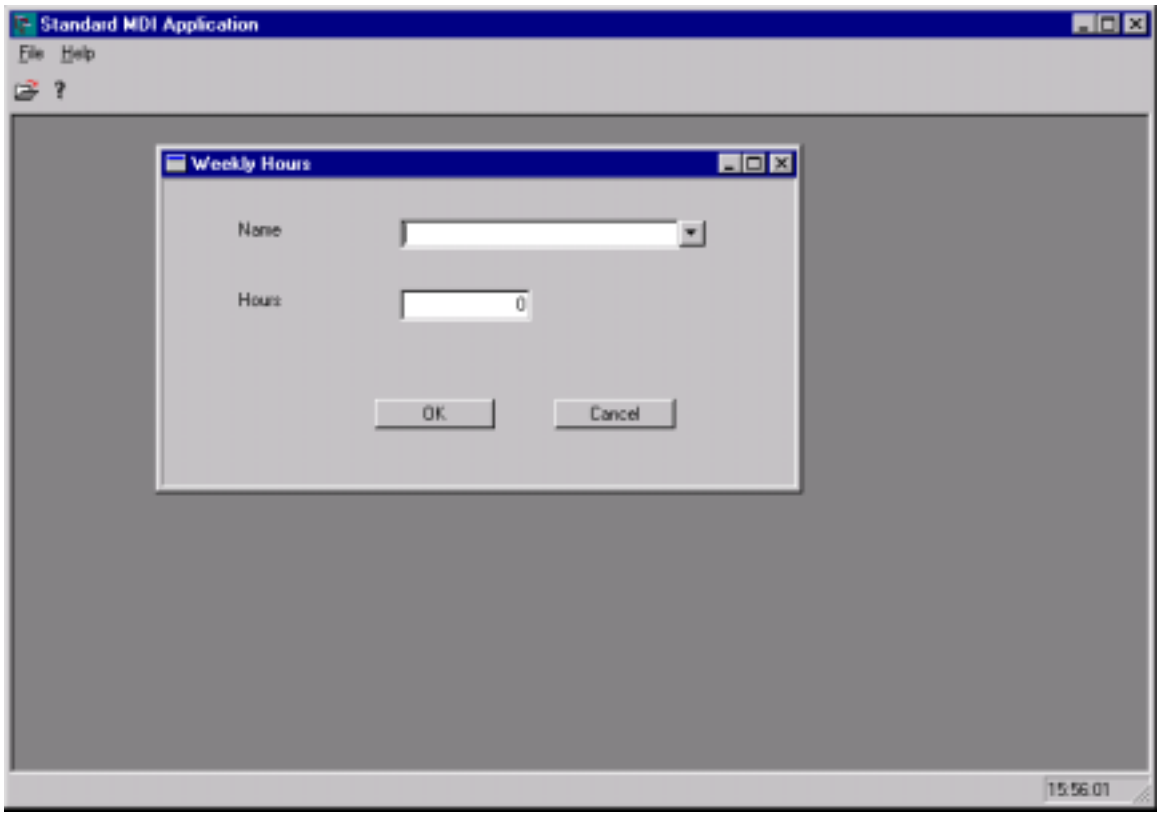
I need a trigger to initiate saving the changes to my Server application, and for the purpose of this demonstration, I will use a push button. This push button will call the Gather method.

```
METHOD Gather() CLASS EmployeeDW
    LOCAL oEmployeeDetails AS EmployeeDetails
    LOCAL dwIndexOrder AS DWORD
    LOCAL sSeekValue AS STRING

    oEmployeeDetails := EmployeeDetails{}
    oEmployeeDetails:FillContent(SELF)

    dwIndexOrder := oSFEmployeeDataBrowserContainer:Server:IndexOrd()
    IF dwIndexOrder == EmployeeNumberIndexOrder
        sSeekValue := oEmployeeDetails:EmployeeNum
    ELSE
        sSeekValue := oEmployeeDetails:Name
    ENDIF

    GetHRServer():SaveEmployeeDWContent(oEmployeeDetails:Content)
    oSFEmployeeDataBrowserContainer:Browser:FillBrowser(sSeekValue)
```

This method makes use of our EmployeeDetails class to transport data between the Client application and the Server application. Just before sending the data off to the Server application, we take note of the next seek value, which we will want to display on our browser. So after control returns from saving the changes, we go back to the Server application a second time and refresh our data browser.

Using Our Browser with a SingleLineEdit

Another function we require from our software is the ability to enter the hours worked each week. We need a screen to identify the employee and then enter the hours. I am going to design a screen similar to the following:

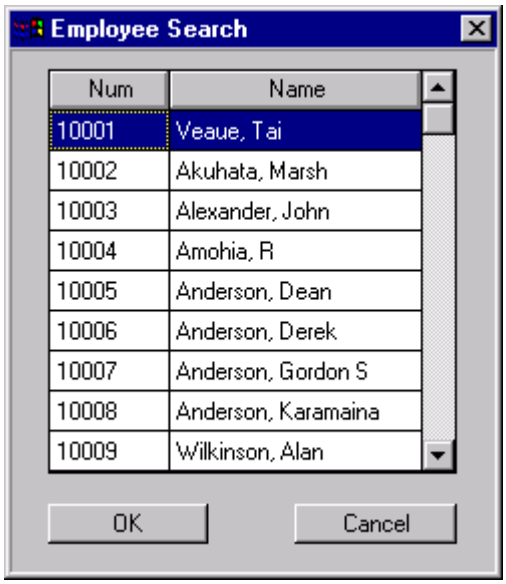


When we are entering the hours and we are prompted for the user's name, we would like to bring up our data browser. The control where we identify the user is a SingleLineEdit control. So two things are required here – we need an Employee Search Dialog window, and we also need to provide some special logic for our Employee Name SingleLineEdit.

Employee Search DataDialog Window

This is the easy part, because all we are going to do is reuse our browser. Create a DataDialog window and call it EmployeeSearchDD. Using the Window Editor, we draw a SubDataWindow called EmployeeDataBrowserContainer and we add two PushButtons – OK and Cancel. We will add some code later on to modify the behavior of the PushButtons.

I used a DataDialog because of its modal properties. When I call up this search screen, the operator is not able to wander to any other part of the application until the screen is disposed of.



SingleLineEdit with a PushButton

This is a class that attaches a PushButton to the right edge of the SingleLineEdit. If the user presses the button, we want some action to occur.

We actually build two objects here, the PushButton object and the SingleLineEdit object.

```
CLASS SlePB INHERIT StdPushButton
   PROTECT oSleWithPB AS SleWithPB

METHOD Dispatch(oEvent) CLASS SlePB
   IF oEvent:Message == WM_LBUTTONUP
      IF SELF:IDisabled
         SELF:oSleWithPB:SetFocus()
      ELSE
         VOSendMessage(SELF:oSleWithPB:Handle(), WM_LBUTTONDBLCLK, 0, 0)
      ENDIF
   ENDIF
RETURN SUPER:Dispatch(oEvent)

METHOD Init(oForm, oResID, oPoint, oDim, oSLE) CLASS SlePB
   SELF:oSleWithPB := oSLE
   SUPER:Init(oForm, oResID, oPoint, oDim)
   SELF:Caption := "..."
RETURN SELF
```

In the Dispatch method above we control what action takes place if the user pushes the PushButton. Note that I am sending a left mouse button double-click message to the SingleLineEdit.

In the SingleLineEdit code we have to calculate the size of the button and then shorten the width of the SingleLineEdit accordingly. The Init method handles all this. The reason for doing it is purely cosmetic.

The only other important feature in this code is ensuring that if the Enable, Disable, Hide or Show method is called, the corresponding method for the PushButton is also called.

```
CLASS SleWithPB INHERIT StdSingleLineEdit
   PROTECT oSlePB AS SlePB

METHOD Disable() CLASS SleWithPB
   SUPER:Disable()
   SELF:oSlePB:Disable()
RETURN NIL
```

```
METHOD Enable() CLASS SleWithPB
   SUPER:Enable()
   SELF:oSlePB:Enable()
RETURN NIL

METHOD Hide() CLASS SleWithPB
   SUPER:Hide()
   SELF:oSlePB:Hide()
RETURN NIL

METHOD Init(oForm, oResID, oPoint, oDim, kStyle) CLASS SleWithPB
   LOCAL lpRect IS _WINRECT
   LOCAL oPBDim AS Dimension
   LOCAL nPBWidth AS DWORD

   SUPER:Init(oForm, oResID, oPoint, oDim, kStyle)
   nPBWidth := SELF:Size:Height - 3
   oPBDim := Dimension{nPBWidth,  nPBWidth}

   SetWindowPos(SELF:Handle(), 0L, 0, 0, SELF:Size:Width - nPBWidth -1, ;
                SELF:Size:Height, SWP_NOMOVE + SWP_NOZORDER)
   GetWindowRect(SELF:Handle(), @lpRect)
   oPoint := SELF:Origin
   oDim := SELF:Size
   oPoint:y += 2
   oPoint:x := oPoint:x + oDim:Width

   oSlePB := SlePB{oForm, 10101, oPoint, oPBDim, SELF}
   oSlePB:Show()

   SELF:OverWrite := OVERWRITE_ALWAYS
   RETURN SELF

ACCESS PushButton CLASS SleWithPB
RETURN oSlePB

METHOD Show() CLASS SleWithPB
   SUPER:Show()
   SELF:oSlePB:Show()
RETURN NIL
```

And there we have a SingleLineEdit with PushButton class, which can be reused as desired.

The Employee Name SingleLineEdit

Our EmployeeSle is going to have a PushButton on it so that we can call up our search screen.

```
CLASS EmployeeSle INHERIT SleWithPB
    HIDDEN lKeysWherePressed AS LOGIC        // Flag if user entered data in SLE
    HIDDEN sEmployeeName, sEmployeeNumber AS STRING
    …

METHOD Init(oOwner, nID, oPoint, oDimension, kStyle) CLASS EmployeeSle
SUPER:Init(oOwner, nID, oPoint, oDimension, kStyle)
SELF:lKeysWherePressed := FALSE
```

As I noted before, when the operator clicked on the button to call up the search screen, the Dispatch method sent a left mouse button double-click message to the SleWithPB. I am intentionally going to avoid the MouseButtonDoubleClick method and instead capture the action in the Dispatch.

```
METHOD Dispatch(oEvent) CLASS EmployeeSle
    …
    ELSEIF  oEvent:Message == WM_LBUTTONDBLCLK
        IF !SELF:ReadOnly
            Send(SELF, #DisplayEmployeeSearchDialog)
            SELF:Selection := Selection{0 ,-1}
        ENDIF
….
```

The Dispatch method calls DisplayEmployeeSearchDialog.

```
METHOD DisplayEmployeeSearchDialog() AS VOID PASCAL CLASS EmployeeSle
    LOCAL oSearchDD AS EmployeeSearchDD

    oSearchDD := EmployeeSearchDD{SELF:Owner}
    oSearchDD:Show()

    IF oSearchDD:lNewEmployeeWasSelected
        SELF:EmployeeNumber := oSearchDD:EmployeeNumberSelected
        SELF:EmployeeName := oSearchDD:EmployeeNameSelected
    ENDIF
SELF:lKeysWherePressed := FALSE
```

Because the EmployeeSearchDD is modal, control moves to the modal window immediately after the Show method call. When the operator closes the modal window, the application continues from the next line immediately after the Show method. To close the modal window, the operator will press the OK or the Cancel button.

```
METHOD OKPB() CLASS EmployeeSearchDD
    SELF:INewEmployeeWasSelected := TRUE
    SELF:EmployeeNameSelected := SELF:SelectedEmployeeName
    SELF:EmployeeNumberSelected := SELF:SelectedEmployeeNumber
    SELF:EndWindow()

METHOD CancelPB() CLASS EmployeeSearchDD
    SELF:INewEmployeeWasSelected := FALSE
    SELF:EndWindow()
```

If the operator presses OK, then in the DisplayEmployeeSearchDialog method above, immediately after the Show method, we store the results of our search. If you go back to the Init method of our SingleLineEdit, which was declared above, I declared two variables:

```
HIDDEN sEmployeeName, sEmployeeNumber AS STRING
```

These variables store the two important parts of the SingleLineEdit display, namely the Employee Number and the Employee Name. If you look at our SingleLineEdit class source code (which will be provided), you will see that this is the only time we store the number and name.

Immediately thereafter, we update the display:

```
METHOD AssignSleValue() PASCAL CLASS EmployeeSle
    SELF:AssignSleValue2(SELF:sEmployeeNumber, SELF:sEmployeeName)

METHOD AssignSleValue2(sNum AS STRING, sName AS STRING) PASCAL ;
        CLASS EmployeeSle
    LOCAL sTextValue AS STRING

    IF Empty(sNum)
        sTextValue := ""
    ELSE
        sTextValue := sNum + "  -  " + sName
    ENDIF
    SELF:TextValue := sTextValue
```

So the Employee Number and Name are formatted and assigned as the SingleLineEdit TextValue.

KeyUp and Dispatch

The only valid contents of this SingleLineEdit are an Employee Number and Name from our Employee databases, so we want to ensure that a valid answer is entered. We therefore want to encourage the user to select a name from our data browser because our data browser contains valid information.

We are going to achieve this by writing a KeyUp and a Dispatch method.

```
METHOD KeyUp(oKeyEvent) CLASS EmployeeSle
   LOCAL nKeyCode AS INT

   SUPER:KeyUp(oKeyEvent)

   IF !IsNil(oKeyEvent:ASCIIChar)
      nKeyCode := oKeyEvent:KeyCode

      DO CASE
   CASE nKeyCode >= LONGINT(Asc(String2Psz("0"))) .AND. ;
           nKeyCode  <= LONGINT(Asc(String2Psz("9")))
      SELF:lKeysWherePressed := TRUE

      CASE nKeyCode >= LONGINT(Asc(String2Psz("A"))) .AND. ;
              nKeyCode  <= LONGINT(Asc(String2Psz("Z")))
         SELF:lKeysWherePressed := TRUE

         CASE nKeyCode >= VK_NUMPAD0 .AND. nKeyCode  <= VK_NUMPAD9
            SELF:lKeysWherePressed := TRUE

      CASE  nKeyCode == VK_DELETE          // or BackSpace
            SELF:EmployeeNumber := ""
            SELF:EmployeeName := ""

      CASE  SELF:lKeysWherePressed .AND. ;  // The keys can be pushed only if
               (nKeyCode == VK_BACK .OR.;    // the user is typing in a new search
                nKeyCode == VK_END .OR.;
                nKeyCode == VK_HOME .OR.;
                nKeyCode == VK_LEFT .OR.;
                nKeyCode == VK_RIGHT .OR.;
                nKeyCode == VK_SHIFT .OR.;
                nKeyCode == 188 .OR. ;            // Comma
                nKeyCode == VK_SPACE)
```

```
            OTHERWISE
                SELF:Selection := Selection{0 ,-1}

        ENDCASE
    ENDIF
RETURN NIL
```

The KeyUp method is designed to keep track of what keys have been pushed. If the operator keys in a number or an alpha key, we set the **lKeysWherePressed** variable to TRUE. If the variable is TRUE, then we also allow the user to press the cursor keys, like Home, End, comma, etc. If I have missed any, this is where you would add them.

So this KeyUp method, basically records if the user has pressed any keys.

```
METHOD Dispatch(oEvent) CLASS EmployeeSle
    LOCAL nReturn := 1L AS LONGINT
    IF oEvent:Message == WM_GETDLGCODE .AND. oEvent:lParam <> 0
        DO CASE
            CASE SELF:lKeysWherePressed .AND. (oEvent:wParam == VK_TAB .OR. ;
                    oEvent:wParam == VK_RETURN)
                Send(SELF, #DisplayEmployeeSearchDialog)
OTHERWISE
    nReturn := SUPER:Dispatch(oEvent)
    ENDCASE
    ELSE
        nReturn := SUPER:Dispatch(oEvent)
    ENDIF
RETURN nReturn
```

The Dispatch method is looking for when the operator uses the Tab key to move to the next control. If it is detected that the user pressed some keys, and then the user presses the Tab key, it is assumed that we have not yet identified the correct employee. So before moving on to the next control, we want to bring up our EmployeeSearch dialog and force the user to select the correct employee.

For cosmetic reasons, I also had the same action occur if the Enter key was pressed.

Now it could be argued here that the operator can click away from the field using the mouse or press the Cancel key on the search screen, and you would be right. After giving this some consideration, I have decided that I should allow the user to leave the SingleLineEdit, but before leaving the control, ensure that I force the SingleLineEdit TextValue to display what I believe to be the last set of keystrokes.

```
METHOD FocusChange(oFocusChangeEvent) CLASS EmployeeSle
    LOCAL lGotFocus AS LOGIC
    lGotFocus := IIf(oFocusChangeEvent == NULL_OBJECT, FALSE, ;
                     oFocusChangeEvent:GotFocus)
    SUPER:FocusChange(oFocusChangeEvent)
    //Put your changes here
    IF !lGotFocus
        SELF:AssignSleValue()
    ENDIF
RETURN NIL
```

The FocusChange method assigns the value for Employee Number and Employee Name into the SingleLineEdit, and if you remember, the only time you get the opportunity to record these values is by displaying and selecting the Search Screen. If they have not selected anything, then the SingleLineEdit is blank.

Streamlining the Search Process

An important aspect of a DCOM application is minimizing the number of visits required to the server. In many cases, this could be the slowest part of your application. If we were to test our application as it is at the moment, we would find that each time we bring up the Search Screen, no trip is made to the server. This is because all we have requested is that it bring up the contents of the server as it stands.

However, I want to make the searching process as streamlined as possible, so I am going to give our SingleLineEdit a bit more intelligence. I am going to achieve this by allowing the operator to key a name into the SingleLineEdit, and then when they request the Search Window, it will open up with the closest match on screen.

The modifications for this need to be made in DisplayEmployeeSearchDialog, where I establish the SearchValue and then do the search before displaying the screen.

```
METHOD DisplayEmployeeSearchDialog() AS VOID PASCAL CLASS EmployeeSle
    LOCAL oSearchDD AS EmployeeSearchDD
    LOCAL sSearchValue

    IF lKeysWherePressed
        sSearchValue := Trim(SELF:CurrentText)
    ELSE
        sSearchValue := SELF:sEmployeeNumber
    ENDIF

    oSearchDD := EmployeeSearchDD{SELF:Owner}
    oSearchDD:FillBrowser(sSearchValue)
    oSearchDD:Show()
```

Where to Position Our Search Screen

When the program brings up the Search screen, we need to control where the window should be placed. Again, I intend to do this in the DisplayEmployeeSearchDialog method.

```
METHOD DisplayEmployeeSearchDialog() AS VOID PASCAL CLASS EmployeeSle
    ...
    oSearchDD := EmployeeSearchDD{SELF:Owner}
    oSearchDD:Origin := SELF:CalculateSearchWindowOrigin(oSearchDD)
    oSearchDD:FillBrowser(sSearchValue)
    oSearchDD:Show(SHOWNORMAL)
```

Here I took the opportunity to line up the Search window with my SingleLineEdit, sitting immediately below the SingleLineEdit. Note, however, that I had to add a parameter to Show method call, which is a constant that represents how the data window is shown. The documentation says that this is the default parameter, so in theory it should not be needed.

Saving the Data to the Server Application

When we have completed filling in our Hours window with the name and the number of hours worked, we then want to save the hours in our DataServer. I have not supplied the complete code to do this, but I do want to point out some important issues here.

If you check the Common HR Library, I have supplied my container for transporting the result to the Server application. Also, I have put the required code in the OKPB method as follows:

```
METHOD OKPB() CLASS EmployeeHoursDW
    LOCAL oEmployeeHoursContainer AS EmployeeHoursContainer

    oEmployeeHoursContainer := EmployeeHoursContainer{}
    oEmployeeHoursContainer:EmployNum := SELF:oDCNameSle:EmployeeNumber
    oEmployeeHoursContainer:Hours := SELF:oDCHoursWorked:Value
    // GetHRServer():SaveEmployeeHours(oEmployeeHoursContainer:Content)

    SELF:EndWindow()
```

What I want to bring to your attention here is that we initialize our container and then we store the Employee Number and the Hours. We do not need the name – it is not required. We only stored it because we needed it to update the SingleLineEdit TextValue. When it comes to storing the hours for the employee, all we need is the Employee Number and the Hours.

Conclusion

When we were first introduced to object orientation, we were introduced to new words like encapsulation (the ability to hide implementation details from the outside world) and Inheritance (the ability to take a class you already have and reuse part or all of it to make a new class). In my opinion, inheritance is where object orientation really shines.

Today we have taken a browser window and given it some intelligence as to how it populates itself, how it can interface with the operator to find a specific record, and lastly how to change the index order.

We then inserted this class into an Employee Maintenance screen, and it worked for us without further instructions. The only programming we did on our Maintenance screen was specific to that screen, not to the browser.

Then as another example, we designed a SingleLineEdit with a PushButton, inherited from that and created our EmployeeSle. If we pushed the button, then a search screen came into view. Designing our search screen involved creating a data window with some push buttons, then placing our data browser into the screen, and done!

I hope you have enjoyed this as much as I have.

Allan Wissing heads an organization called Computer Craft, a New Zealand based company, whose main focus is software development. The company has been providing software support, consulting, analysis, and programming services since 1983. In 1985 Allan began developing Clipper applications and moved on to CA-Visual Objects when it was released. For the past 13 years, Allan has consulted for many of New Zealand Fortune 50 companies, and developed ISM product, which is an Industrial Health & Safety Management application. Allan can be reached by email at Allan.Wissing@Computercraft.co.nz.