Chapter 5
Creating and Using Windows

Objective

This lesson covers the various types and styles of windows that are available in CA-Visual Objects. The first objective is to understand the difference between a Multiple Document Interface (MDI) application and a Single Document Interface (SDI) application.

Secondly, two of the most widely used types of windows—modal dialog windows and MDI client windows—are discussed. Modal dialog windows are commonly used in Windows applications when a user response is essential before the program can continue. An MDI client window (or child application window) provides the added benefit of allowing you to add data-aware behavior to your window.

In this lesson, you will learn how to:

- Create an SDI application, using the TopAppWindow class
- Create a MDI client window, using the ChildAppWindow class
- Create a modal dialog window, using the DialogWindow class
- Add data-aware behavior to a ChildAppWindow, using the DataWindow class

Overview

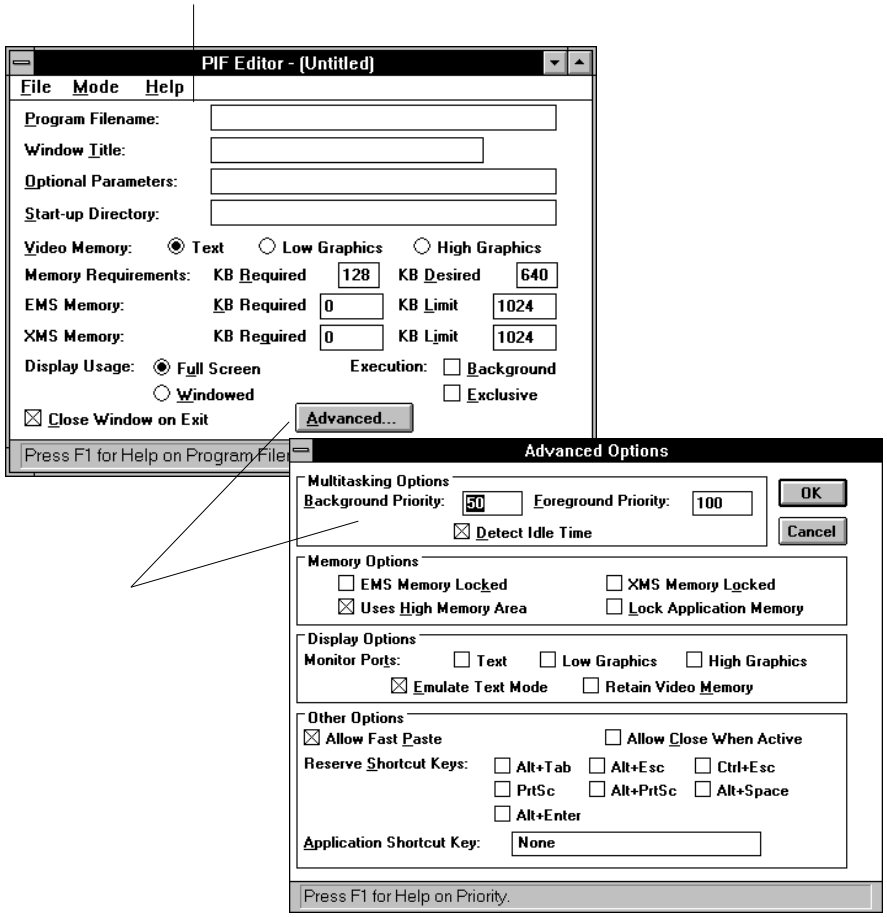
CA-Visual Objects allows you to create both Single Document Interface and Multiple Document Interface applications with several types of windows based on subclassing the various Window classes.

Single Document Interface Applications

The Single Document Interface (SDI) is a user-interface standard for presenting and manipulating a single document within a Windows application. An SDI application has one main window in which the user can open and work with a single document. SDI allows for a more classical (linear) approach to application interface.

An SDI application can spawn a child window. However, when the main window is closed, so is the child. The child window can move outside the main application window, but it is bound to the document held in the main application window.

The Windows PIF Editor is a good example of an SDI application, in that a child window appears when you click on the Advanced push button:



Top Application Windows

As you have just seen in the PIF Editor example, a top application window is the main window of an SDI application. It has no owner windows. As with other application windows, a top application window can have icons, captions, resizable borders, menus, and system menus.

An application, however, can have *more* than one top application window. From a user's standpoint, multiple top application windows would appear as multiple applications. To a programmer, multiple top application windows could readily share information and interaction.

The following example creates and invokes two top application SDI-style windows, using the TopAppWindow class:

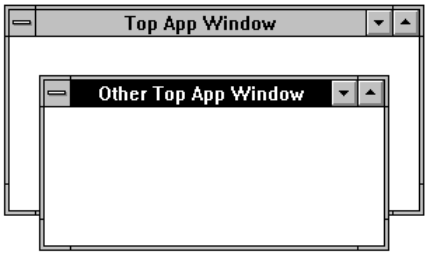
```
METHOD Start() CLASS App
    LOCAL oTopAppWin AS TopAppWindow
    LOCAL oWin2 AS TopAppWindow

    oTopAppWin := TopAppWindow{}
    // Create a top application window
    oTopAppWin:Caption := "Top App Window"
    // Assign new window caption
    oTopAppWin:Origin := Point{150,150}
    // Pixels from bottom left of screen
    oTopAppWin:Size := Dimension{400,250}
    // {width,height} of new window in pixels
    oTopAppWin:Show()
    // Show new window

    oWin2 := TopAppWindow{}
    oWin2:Caption := "Other Top App Window"
    oWin2:Origin := Point{100,100}
    oWin2:Size := Dimension{300,300}
    oWin2:Show()

    SELF:Exec()
     // Windows Exec() loop
```

This code would create the following:



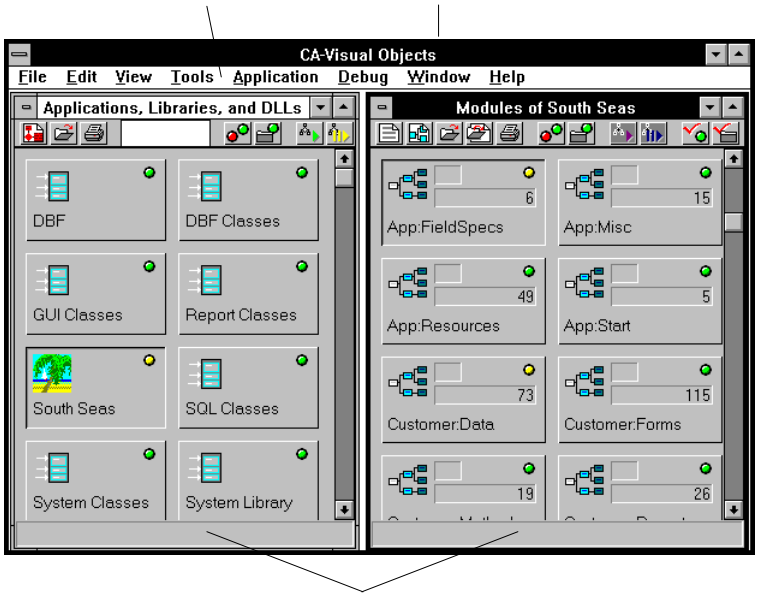
This application has two independent SDI windows that can perform system functions (such as move, size, minimize, and so on). You would use a TopAppWindow window as the top window in any SDI application.

However, SDI applications are not as common as MDI applications because the Multiple Document Interface handles multiple tasks more robustly, as you will see next.

Multiple Document Interface Applications

The multiple document interface (MDI) is a user-interface standard for presenting and manipulating multiple documents within a single Windows application. An MDI application has one main window, in which the user can open and work with several documents (for example, text files, databases, or spreadsheets). Each document appears in its own child window inside the main application window.

The CA-Visual Objects desktop is a good example of an MDI application:



Because each child window has a frame, system menu, Maximize and Minimize buttons, and an icon, the user can control it just as if it were a normal, independent window. Child windows however, cannot move outside the main application window.

Tip: As a general rule, whenever you see the Tile or Cascade menu commands under the Window pull-down menu, then you know you are dealing with an MDI application.

Shell Windows

A shell window is similar to a top application window except that it acts as an MDI parent window. All children of this window act as MDI child windows. Most of the applications you write will be MDI applications, simply because they are adept at managing multiple tasks.

Dialog Windows

Dialog windows (or *dialog boxes*) are used to present and gather information. They can return a result that indicates user interaction (for example, if the user pressed the OK or Cancel push button). There are two types of dialog windows—modal and modeless.

Dialog windows are generally used to present specific questions to users and accept their responses; hence, they are generally modal. CA-Visual Objects allows both modal and modeless dialog windows.

Modal Dialog Windows

Modal dialog windows must be acknowledged before the current thread of execution can continue. A further distinction must be made between system modal and application modal dialogs windows:

- Dialog windows that are system modal must be acknowledged (by a button click for instance), before any execution by any currently running applications, including the Windows desktop, can continue.
- Dialog windows that are application modal stop only the current application thread. The user is able to use the Alt+Tab keystroke to jump to another application.

Dialog windows can be created as modal, by specifying the third parameter of the DialogWindow:Init() method as TRUE.

Modeless Dialog Windows

Modeless dialog windows, on the other hand, do not affect the current execution thread. Although not used very often, a modeless dialog window can be useful as a progress bar indicator, or a search and replace routine in a text editor.

Dialog windows can be created as modeless, by specifying the third parameter of the DialogWindow:Init() method as FALSE.

Child Application Windows

A child application window is an application window that "belongs to" another window (its owner). Because a child application window is not independent of the owner window, it is always destroyed, hidden, or iconized when its owner window is destroyed, hidden, or iconized. Also, child application windows are never displayed outside of the boundaries of the owner window.

Child application windows do not have a default size and position on their owner window. They must be assigned an origin and size before they are displayed.

One way to create a ChildAppWindow object on an MDI shell window is as follows:

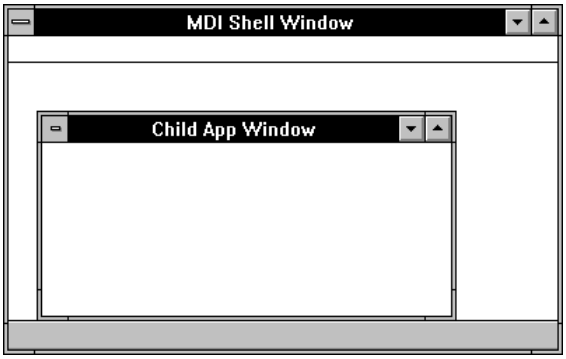
```
METHOD Start() CLASS App
       LOCAL oShellWin AS ShellWindow
       LOCAL oChildAppWin AS ChildAppWindow

       oShellWin := ShellWindow{}
       oShellWin:Caption := "MDI Shell Window"
       oShellWin:Origin := Point{150,150}
       oShellWin:Size := Dimension{400,250}
       oShellWin:Show()

       oChildAppWin := ChildAppWindow{oShellWin}
       // Create child window owned by oShellWin
       oChildAppWin:Origin := Point{10,10}
       // Relative to oShellWin
       oChildAppWin:Size := Dimension{300,150}
       oChildAppWin:Caption := "Child App Window"
       oChildAppWin:EnableBorder()
       // This allows resizing of the child window
       oChildAppWin:EnableSystemMenu()
       // Turns on the system menu
       oChildAppWin:Show()

       SELF:Exec()
```

When executed, this code produces the following:



Using the Window Editor

In most cases, you will not write code to create your windows. Data windows, in particular, are extremely complex. It is much more convenient to use the Window Editor to do this for you.

The standard way to build a window is to design its layout in the Window Editor. This produces a resource file that specifies what controls the window has, along with their locations, sizes, captions, and accelerators. It also generates a window subclass and an Init() method, that associates names and further annotations with each control. With such a redefined layout, the window is displayed very quickly and the methods of the window have enough information to act intelligently.

MDI Windows and Menus

In Windows, a child application window never displays a menu (except for the system menu). However, it can own a menu. If the child window is used as an MDI child under a shell window, the child's menu is displayed in the shell window. This is very useful: for instance, when different child windows present different types of data, they often need different menus. The shell window automatically "knows" to replace its own menu with the menu of the child window that has focus.

The shell window menu itself is used only when:

- There are no open child windows
- The active child window does not have an associated menu

It should be pointed out that this ownership causes some unique behavior. If an event is called from a menu, the application looks to the currently selected window—the window with focus—for a method to invoke (that is, a method of the current window class). If it cannot find one, it looks to the owner window. This allows you to add a method to a shell window, which can easily be invoked from any ChildAppWindow's menu.

For more information on event propagation in menus, see the "Creating Menus and Toolbars" chapter in this guide.

Data Windows

The DataWindow class inherits from the ChildAppWindow class, acquiring its behavior. It also adds data-aware behavior that enables it to interact intelligently with data servers.

When connected to a data server, a data window forms a view of the server that allows for direct access and manipulation of the server's data.

Server Use

A data window is connected to a server via the Use() method. When this connection is established, each edit control on the window is connected to a field in the data server based on matching names: a field named CustName in the server is connected to the control named CustName in the data window. Assigning a value to a control automatically propagates it to the server.

Data Propagation

When a control is connected to a field in a data server, a value entered into the control, or assigned to the appropriate name from the program, is automatically propagated to the server. Thus, after executing this statement,

```
oCustomerWindow:CustName := "Albert Stanley"
```

the CustName field in the server has the correct value. This is referred to as *name-based linkages*.

Values are propagated up from the data server to the data window when the server repositions itself, or when another window makes a change. This requires no special action: after executing a Skip() method or assigning a value to a field, every window connected to the server is automatically updated to reflect the change.

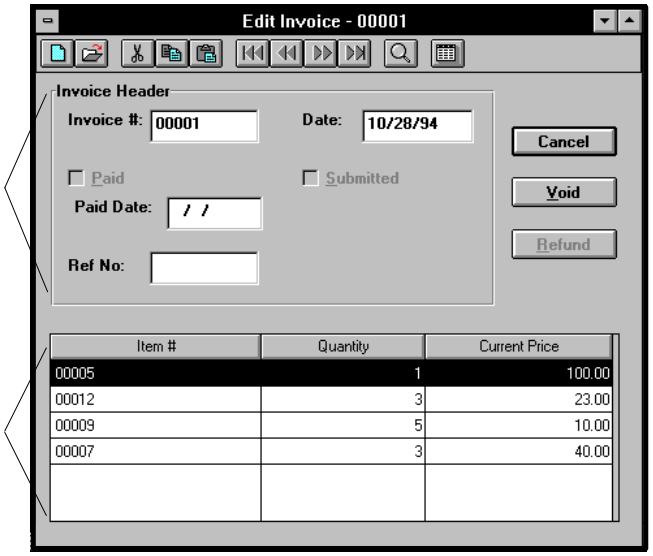
Master Detail Option

Data windows can be associated with servers in two ways: in a single-server or dual-server (master-detail) relationship.

The Master Detail Option in CA-Visual Objects allows you to link a window to two data servers in a master-detail relationship. In this case, a data window is created for the master server with a nested sub-data window for the detail server.

This produces a sub-data window or "subform control." The sub-data window is just another window control (like a push button or single-line edit control), except that it represents a subset of a joined table. One of the fields in the master data server would be used to perform a SetSelectiveRelation() to the detail server based on its controlling order.

For example, the Edit Invoice window of the South Seas Adventures application contains a master-detail relationship:



Only a data window can own a sub-data window because it is the only window type that can deal with data-related events.

Form and Browse View

A data window can take on two different view modes:

- Form view contains individual controls for the data fields for a single record
- Browse view contains a spreadsheet-like data browser for displaying multiple records

The data window can be initially displayed in either mode, and can be switched to the other mode at any time (implemented by using the DataWindow:ViewAs() method). Any data window supports both appearances, although you can, of course, choose not to provide a way to select one mode or the other by deactivating or removing one of the standard menu commands.

The two view modes provide the same set of facilities: the same data linkage facilities, the same display options, and the same data manipulation methods. From the perspective of the application, a data window has the same behavior and the same data properties regardless of view mode.

Data Validation

Data entered by the user is automatically validated using any one of the validation rules for the field specification attached to a control or column. (For more detail on the validation rules provided, see the “Defining Field Specifications” chapter in this guide.) If data fails the validation test, the diagnostic message of the validation rule is displayed on the status bar. The data window does not propagate invalid information down to the server or to other windows, nor does it take any action that requires writing the invalid value to the server.

In addition, CA-Visual Objects allows the specification of a ValidateRecord() method for the window as a whole, for the purpose of cross-field validation. This method, if provided, is called after all the fields have passed their individual validations.

Action Methods

The DataWindow class also provides action methods that correspond to the general capabilities of all data servers: GoTop(), GoTo(), GoBottom(), SkipNext(), SkipPrevious(), Append(), Delete(), and so on. The DataWindow versions of these methods verify the validation status of the controls on the window. If everything is valid, the window invokes the corresponding method of the data server.

Concurrency Control

The concurrency control mode identifies when and how records are locked and released. This mode can be set by assigning a constant to the ConcurrencyControl instance variable of a data window. Because of this instance variable, you do not have to worry about locks in many situations. The available constants that you can set are defined, as follows:

Constant	Description
CCFILE	All the records in the entire set provided by the server are locked throughout. This is not very practical for windows associated with all the records of a server, since it would correspond to a file lock. It is intended to be used in conjunction with method DataWindow:SetSelectiveRelation().
CCNONE	The data window provides no automatic record locking; the application is required to do all locking explicitly.

Continued

Continued

Constant	Description
CCOPTIMISTIC	No locks are maintained continuously. Before any update is done, the record is reread from disk and compared with what was previously read into the buffer. If it matches, the record is locked, the update is done, and the lock is released. If it does not match, a failure is returned and the new data is propagated up to the window. This is the default.
CCREPEATABLE	All records that have been read are maintained locked. The user is guaranteed that when moving back among previously viewed data, they are unchanged.
CCSTABLE	The record that the window is sitting on is always kept locked. Note that when in browse view, the row that the cursor is on represents the current record.

The data window provides built-in facilities for concurrency control, automatically setting and releasing record locks as appropriate. Once the appropriate mode has been selected by assigning one of the available constants to the ConcurrencyControl instance variable, record movements and changes use the concurrency control facilities of the data server to take and release locks. If taking a lock fails because another user controls the record, the data window action fails; the default action of the data window is to display a message in the status bar.

Disconnected Controls

If controls are not linked by name, they essentially become buffers. The data window treats these as ordinary controls and takes no action relative to the attached server. This is often the desired approach when one wishes to perform actions that should be buffered from the server until a user completes all tasks on a window (for example, completing the required fields before creating a new record).

Consider the case where you are using a regular data window with linked controls to do your edits. All validations are being done automatically. Suppose the application has to be able to add new records to the table. The simplest method is to put a button on the edit window that appends a blank record. After appending the record, the user may decide to cancel the update. Now, you have the problem of deleting the blank record.

This is avoided in the South Seas Adventures application by buffering all append operations. In the application, auto-layout is initially used to populate the data window. The names are then changed to disconnect the controls from the fields in the server that they represent. If the user clicks the OK button, then—and only then—does the record append occur.

Exercise

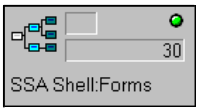
Viewing an MDI Application

Let's take a closer look at an MDI application and its shell window and various child windows.

The Shell Window

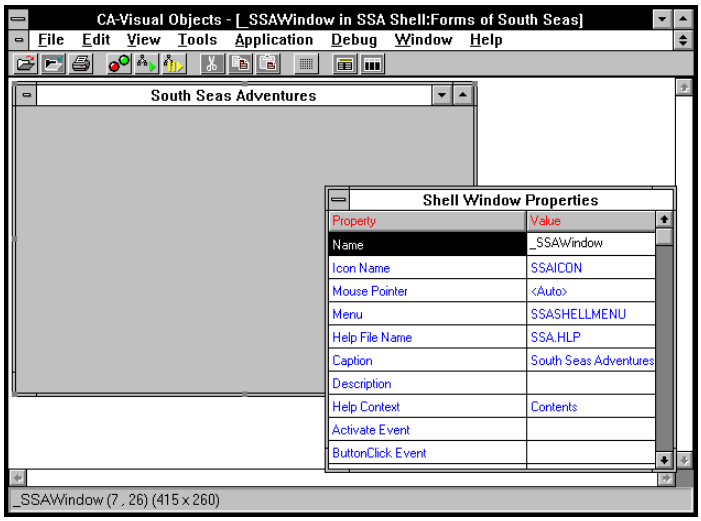
The South Seas Adventures application uses a shell window as its MDI parent window. It was created using the Window Editor. Let's inspect that window now.

1. Open the SSA Shell:Forms module, by double-clicking its module button:



2. Double-click on the window entity called _SSAWindow.

   You are presented with the shell window for the South Seas Adventures application:



Notice that the Shell Window Properties dialog box shows the name of the class as _SSAWindow.

3. Close the Window Editor by double-clicking on its system menu.

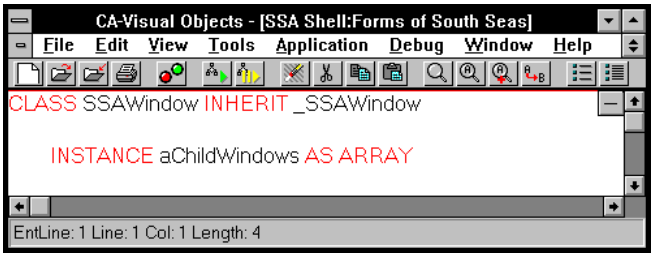
Subclassing Your Windows

What if the windows created by the Window Editor do not do everything that you want them to do? What if you want to add some more functionality? You do not want to throw away the Window Editor and code from scratch. A useful technique is to subclass from the code that the Window Editor generates for you.

In the South Seas Adventures application, there is a need to keep track of all child windows that have been instantiated. A new class called SSAWindow is used to do this; it inherits from _SSAWindow. The SSAWindow class is used to instantiate the South Seas Adventures application shell window. Let's see what is involved.

1. Open the SSAWindow class by double-clicking on its entity button.

   You are presented with a Source Code Editor window:



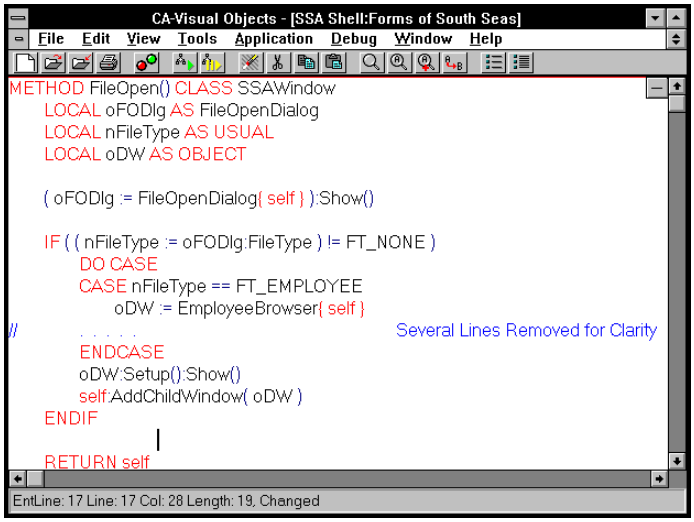
   It shows that an instance variable called aChildWindows has been added to the SSAWindow class. SSAWindow is subclassed from _SSAWindow (which is generated from the Window Editor). A reference to each child window is stored in the aChildWindows array as it is opened.

Let's see how child windows are opened and tracked.

2. Close the Source Code Editor by double-clicking on its system menu.

3. At this point, you should already have the SSA Shell:Forms Entity Browser active. Double-click on the SSAWindow:FileOpen() method.

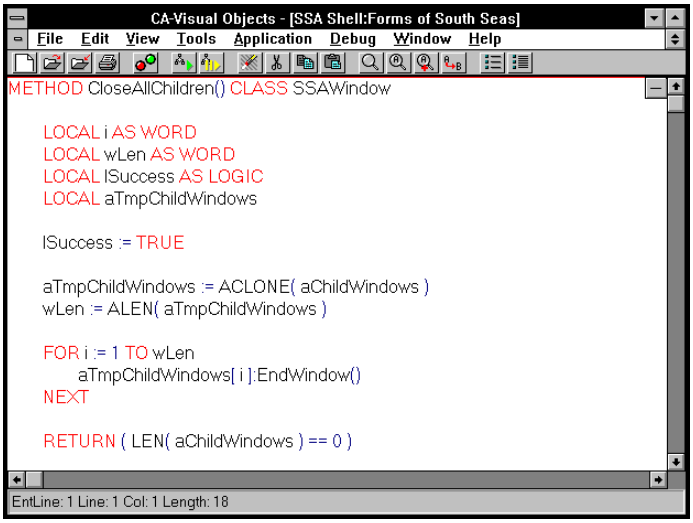
   In the source code for the FileOpen() method, notice the new window being added to the array of windows:



Note: Keeping track of all child windows is a useful technique, since the application can now perform operations against all child windows. For example, South Seas Adventures is designed to allow closing all child windows without closing the application. This is accomplished by the CloseAllChildren() method. You will look at this code next.

4. Close the Source Code Editor and double-click on the SSAWindow:CloseAllChildren() method in the SSA Shell:Forms Entity Browser.

   The code for the CloseAllChildren() method is displayed:



5. Close the Source Code Editor window by double-clicking on its system menu.

Creating a Modal Dialog Box

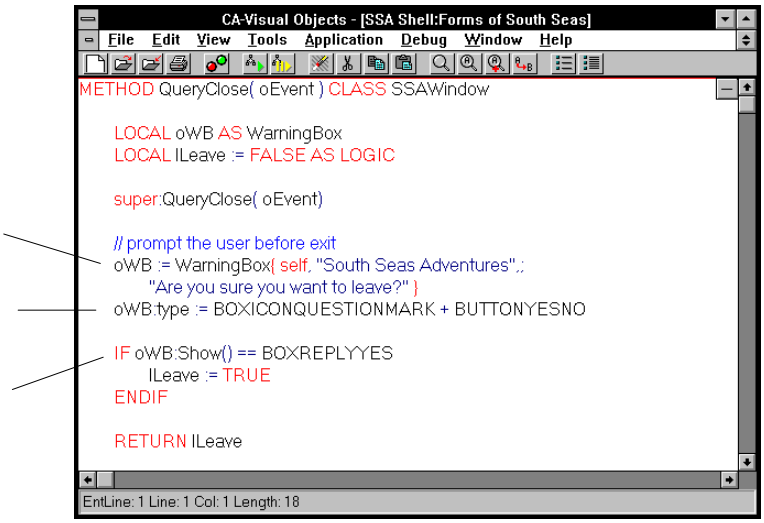
In this exercise, you will create a standard “warning” dialog box that is modal. Typically, a modal dialog box shows your users some information and then returns some value.

Warning Box Modal Dialog Windows

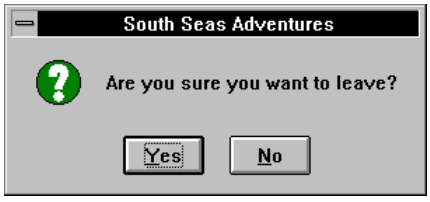
The WarningBox class creates a simple modal dialog box that asks users for verification before an action occurs. Let’s see what is involved in creating one:

1. In the SSA Shell:Forms Module Browser, double-click on the SSAWindow:QueryClose() method.

   The Source Code Editor appears:



   When this code is executed, the warning dialog box looks like the following:



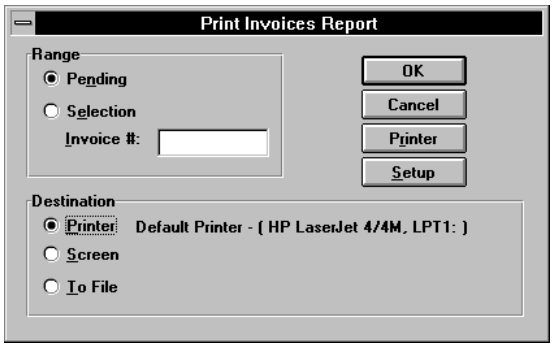
Modal dialog windows actually stop your code by replacing the App:Exec() loop with their own Execute() loop.

2. Close the Source Code Editor by double-clicking on its system menu.
3. Close the Entity Browser for SSA Shell:Forms by double-clicking on its system menu.

Retrieving Values from Modal Dialog Windows

It is pretty simple to get one result back from a dialog box—how about many results?

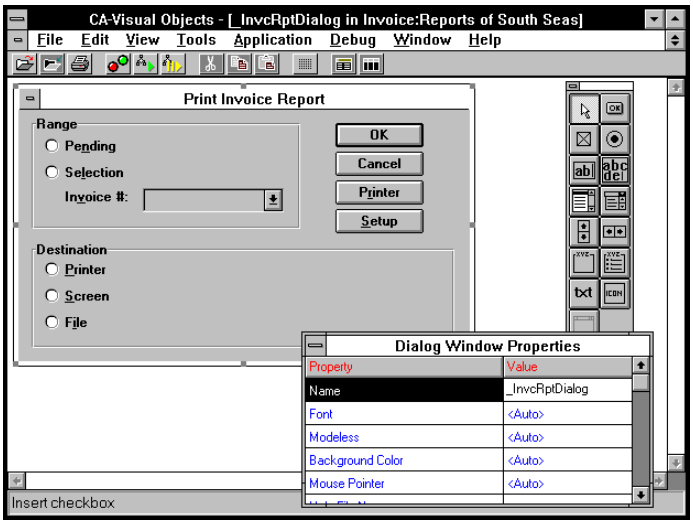
In the South Seas Adventures application, selecting the Invoices command from the Report menu invokes the following dialog box:



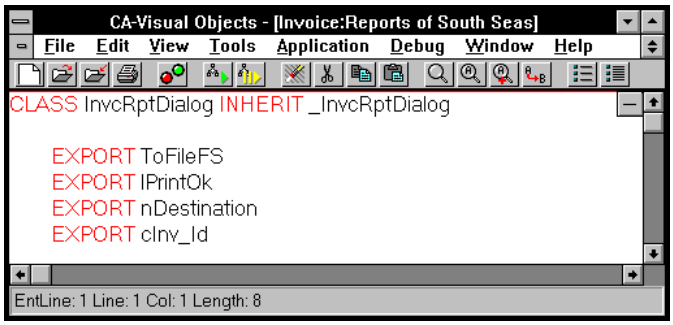
When DialogWindow objects (such as the one shown above) are closed using EndDialog(), their TextControl objects—such as single-line edit controls, check boxes, etc.—are destroyed. This means that the TextControl objects cannot be queried after the user clicks the OK or Cancel buttons.

The method used in the South Seas Adventures application to circumvent this problem is to:

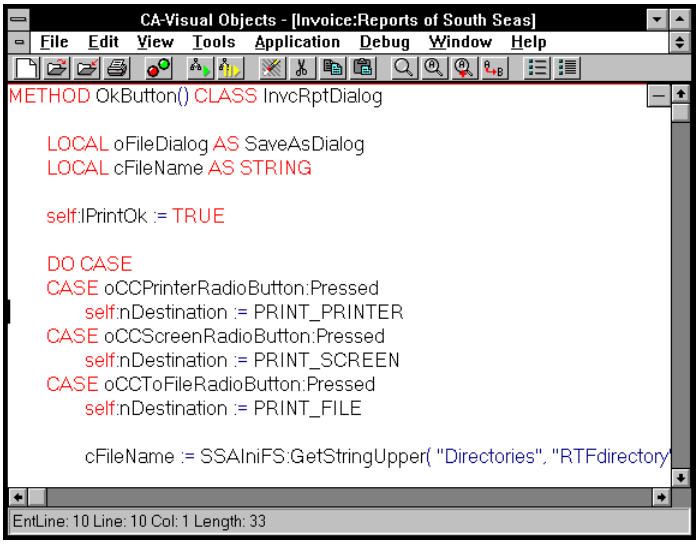
- Use the Window Editor to create a dialog window



- Subclass and create any necessary instance variables



- Update the instance variables when the user clicks OK



This methodology allows you to write code that would query the results of the dialog box as follows:

```
oDialog := InvcRptDialog{SELF}
oDialog:Show()

DO CASE
       CASE oDialog:nDestination == PRINT_PRINTER
              // Send report to printer
       CASE oDialog:nDestination == PRINT_SCREEN
              // Send report to screen
       CASE oDialog:nDestination == PRINT_FILE
              // Send report to file
       OTHERWISE
              // Do nothing
ENDCASE
```

Creating a Data Window

In this exercise, you are going to create a data window to support the editing of customer records. This data window will be attached to the Customer data server. You will use the Auto Layout feature to propagate the data, and then add an OK push button.

Importing a Support Module

The support methods for related push buttons have already been created for you, and are stored in a module export file (.MEF) for you to import:

1. Open the South Seas Adventures application by double-clicking on its button in the Application Browser.
2. Select the Import command from the File menu.
3. From the Import dialog box, select the TUTWIND.MEF file located in the SAMPLES\SSATUTOR\FILES subdirectory which can be found below the CA-Visual Objects installed directory.
4. Choose OK.

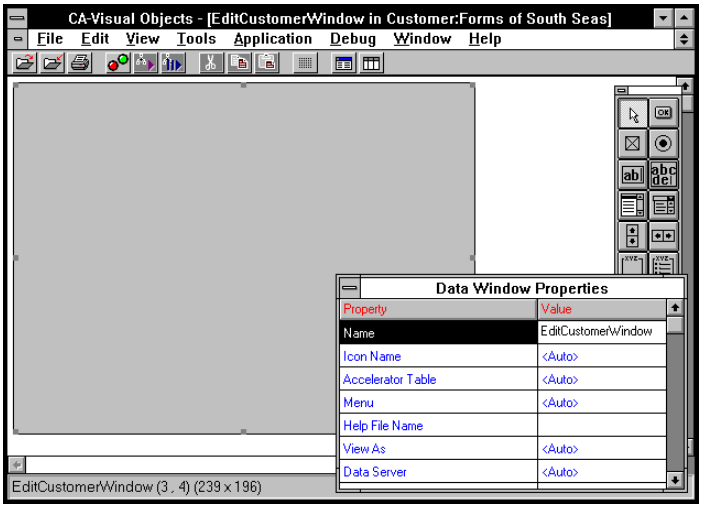
Creating a Data Window Template

1. In the South Seas Adventures Module Browser, select the Customer:Forms module by clicking on its button.
2. Select the Window Editor command from the Tools menu.

   The Window Editor dialog box appears, allowing you to define the type of window (DATAWINDOW is already selected) and its name:

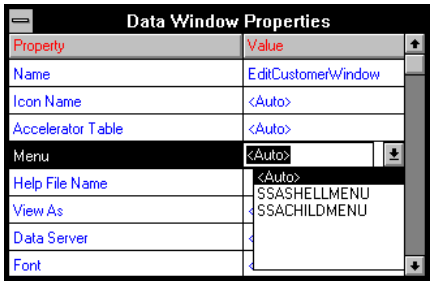
3. Type **EditCustomerWindow** in the Name edit control.
4. Choose OK.

   The Window Editor is displayed:



   The Window Editor provides you with a window template for you to use in designing a data window.

5. In the Data Window Properties box, choose the Menu property and select SSACHILDMENU in the drop-down list box.



This attaches the SSAChildMenu menu to the data window.

6. Choose the Caption property and type **Edit Customer**.

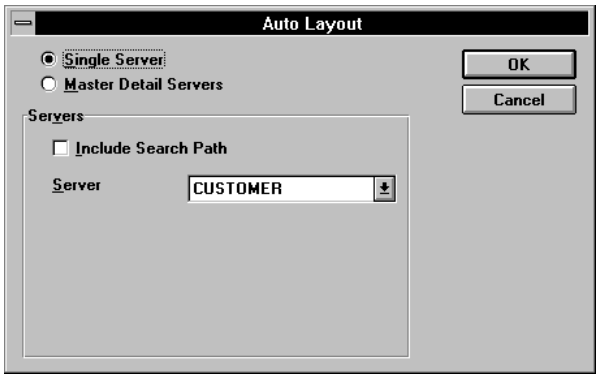
   This caption is used for the title of the data window title.

Designing Your Window Layout

Auto Layout is a convenient way to create and position edit controls on a template window very quickly. Each control corresponds to a data field of the selected data server. It is recommended that you use the Auto Layout feature and then edit and / or move the controls as you require.

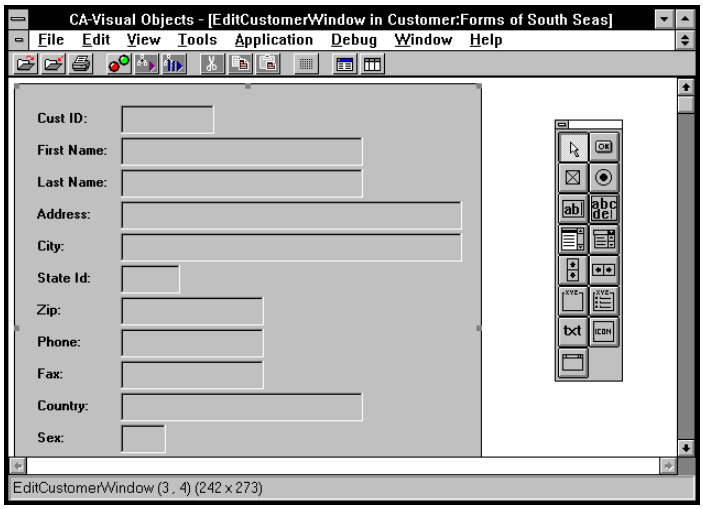
1. Click on the Auto Layout toolbar button.

   The Auto Layout dialog box appears:



2. Select the Single Server option.
3. Select CUSTOMER from the Server drop-down list box.
4. Choose OK.

This automatically lays out all of the fields defined to the Customer data server on the window template, as shown below:

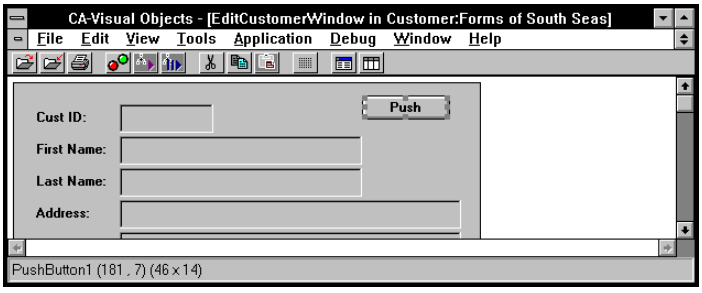


If you want, you can edit, move, or delete any of these fixed text and single-line edit controls from the predefined data window.

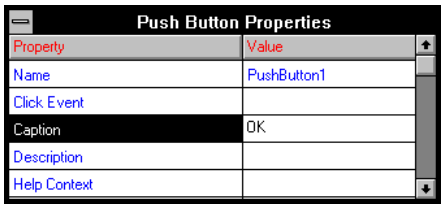
Adding a Push Button

Now let's add a push button to the data window.

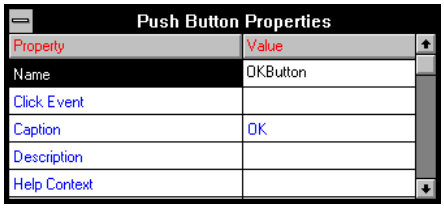
1. Select the Push Button icon from the Window Editor's tool palette.
2. Drop the push button control at the top right-hand corner of the data window template:



3. Change the Caption property to **OK**, using the Push Button Properties Window:



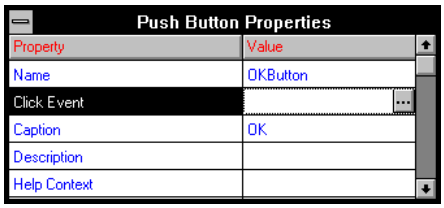
4. Change the Name property to **OKButton**:



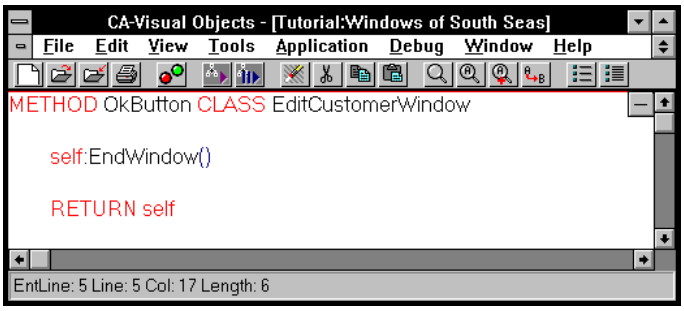
   This conforms to the naming convention used throughout the application.

5. Select the Click Event property and click on the Ellipsis button.

   This allows you to edit the method to be executed when the button is clicked:



6. The EditCustomerWindow:OKButton() method appears in the Source Code Editor:



You do not need to make any changes to the OKButton() method source code.

Compiling and Testing Your Changes

To verify the results of your changes:

1. Close the Source Code Editor window by double-clicking on its system menu.
2. Close the Window Editor by double-clicking on its system menu. Select Yes when prompted to save the changed entities, if you made changes.

3. Build the application by clicking on the Build toolbar button.

4. Run the South Seas Adventures application by clicking on the Execute toolbar button.
5. Log in as usual (Name: **User**, Password: **Trainee**).

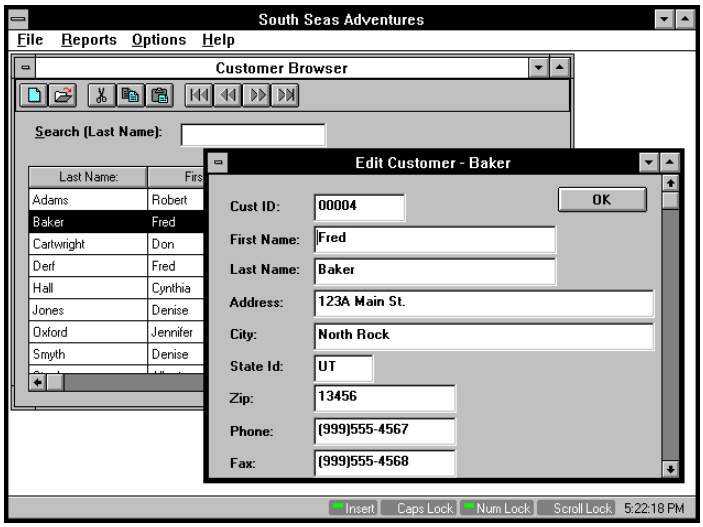
6. Select the Open command from the File menu.

   You are presented with the Open File dialog box:

7. Start a Customer file edit session by clicking on the Customer radio button.

8. Choose OK.

   This opens the Customer Browser window:



9. Click on the name Baker in the Customer Browser window, and then click on the Edit toolbar button to open the newly created Edit Customer window.

10. Choose OK to close the Edit Customer window.
11. When you are finished, exit the South Seas Adventures application by double-clicking on its system menu.

    In the "Adding Controls to Your Windows" chapter of this guide, you will come back to this window to add a Cancel push button.

Summary

This lesson covered a lot of information regarding window creation. You should now understand the difference between SDI and MDI applications and have a good understanding of the most commonly used window types, including:

- Shell windows
- Dialog windows
- Data windows

You should also know how to use the Window Editor Auto Layout feature to quickly create an application editing window.

In the next lesson, you will discover the various controls that you can use to customize your windows—including radio buttons, check boxes, and list boxes.