Chapter 10
Customizing Window Event Handlers

Objective

This lesson discusses window objects and window event processing. In this lesson, you will:

- Examine several events that are often customized in CA-Visual Objects applications
- Customize your own window event handlers
- Define how windows respond to different types of event messages

Overview

Events and Event-Driven Programs

In Windows, actions—such as pressing a key and clicking a mouse—are referred to as *events*. Applications written for the Windows environment respond to these events, thus Windows applications are considered to be *event-driven*.

A program performs a single task for each event it receives. Routines must be supplied for each event. Because the order in which events occur is unpredictable, these routines must be self-sustaining. That is, the routine must be able to respond to the event with little or no knowledge of what events may already have occurred or which events may be pending. This may be seem like an impossible task, but since the events are very well detailed, the job can be managed. This provides a flexible and powerful programming interface.

In Windows, many events are generated as a result of user interaction with your program. Key presses, mouse clicks, resizing a window, and selecting a control are all events generated by the user.

Windows and Events

Windows monitors all events in the environment and is responsible for placing relevant event messages in a message queue for your application. The GUI classes retrieve messages from the queue and dispatch them to the appropriate routine in your application—an *event handler*—for handling a particular event.

There is an inherent compatibility between event-driven programming and object orientation. In an object-oriented program, messages are sent to objects to communicate with them. In event-driven programs, event messages are dispatched to application objects to notify them of an event. In each case, how the message is handled is up to the object. This natural similarity makes object-orientation an excellent framework for developing event-driven systems.

CA-Visual Objects and Events

CA-Visual Objects takes full advantage of this framework. When the CA-Visual Objects dispatcher receives an event message from Windows, it creates an event object. The dispatcher then sends a message (invokes a method) to the appropriate window object.

The CA-Visual Objects Window classes inherit from the EventContext class. It is through Window classes that events are propagated. They are equipped with the necessary event-handler methods, all of which provide default behavior.

Events and Your Application

Although CA-Visual Objects provides default behavior, it is through customizing these event-handler methods that your application can really stand out in a crowd.

The event handlers provided by CA-Visual Objects are encapsulated as methods of the Window class hierarchy. By subclassing the appropriate window method, you can override or enhance the default behavior.

In this lesson, you will examine the following list of events:

Event Name	Occurs When
ButtonClick	There is a mouse click on a push button, radio button, or check box.
EditChange	An edit control (such as a single or multiline edit control) is modified.
Expose	Part of the window needs repainting.
ListBoxSelect	An item in a list box control (list box or combo box) is selected.
Notify	When a data window's attached data server sends a notification.
QueryClose	A window is about to be closed.

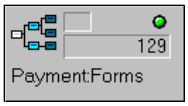
For more information on event handlers, refer to the "Classes and Methods" topic in the CA-Visual Objects online Help.

Exercise

Let's now take a look at the event methods that are handled within the NewPaymentWindow window in the South Seas Adventures application:

1. Open the South Seas Adventures by double-clicking on its button in the Application Browser.

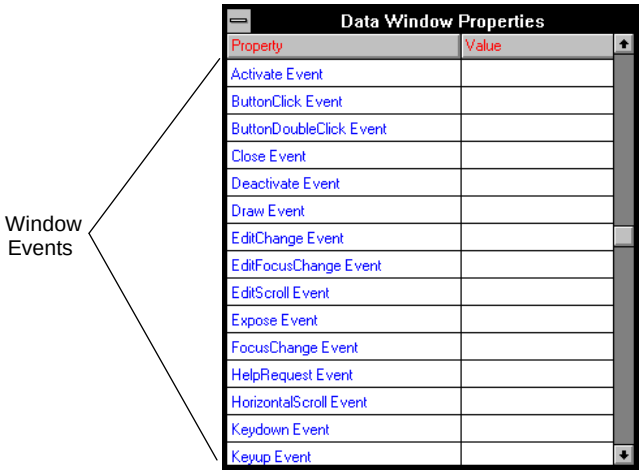
2. Open the Payment:Forms module by double-clicking on its button in the Module Browser.



3. Open the NewPaymentWindow window by double-clicking on this entity in the Entity Browser.

   The Window Editor is displayed.

4. In the Data Window Properties window, scroll through the events that are handled by the window (ActivateEvent, ButtonClick Event, and so on).



   If you want to customize the behavior of a window, just redefine the event methods that you wish to change—it's that simple.

ButtonClick, EditChange, and ListBoxSelect Events

A ButtonClick event occurs when any button control (radio button, push button, or check box) is clicked.

An EditChange event occurs when any edit control (single or multiple line) is modified.

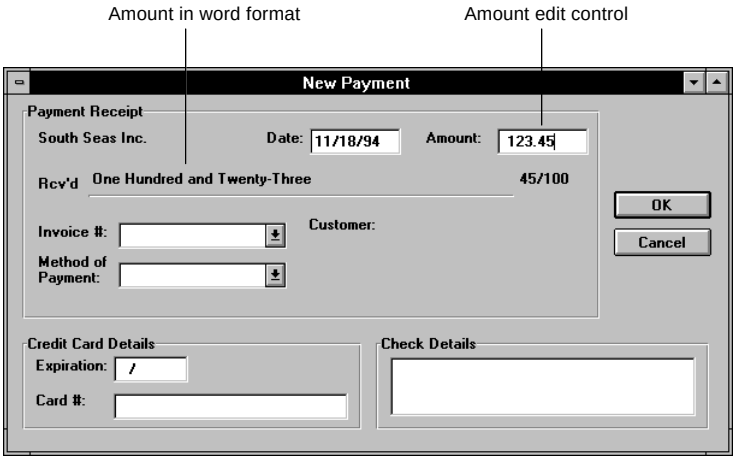
A ListBoxSelect event occurs when an item in any list box control (list box or combo box) is selected.

These events are all similar in nature, taking place when controls are modified. Each of these events also share the appropriate event handler method for all controls of the same type.

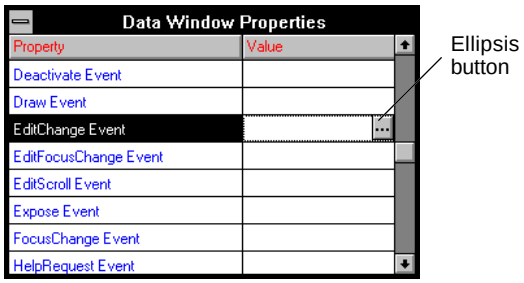
Creating an EditChange() Method

Let's add an EditChange() method to the NewPaymentWindow window.

The new method prints the amount in word format across the payment receipt as the amount is entered in the amount single-line edit control.



1. Click on the EditChange Event property in the Data Window Properties window, then click on the ellipsis button.



This automatically launches the Source Code Editor. It also supplies you with the base source code for your method:

```
METHOD EditChange(oCE) CLASS NewPaymentWindow
    LOCAL oC := oCE:Control
    LOCAL uValue
    SUPER:EditChange(oCE)
    // Put your changes here
RETURN
```

2. Now, modify the EditChange() method as follows:

```
METHOD EditChange(oCE) CLASS NewPaymentWindow
   LOCAL oC := oCE:Control
   LOCAL uValue
   SUPER:EditChange(oCE)
   // Put your changes here

   IF oC:NameSym == #MAmount

     oDCDollarsText:Value :=;
          CWhole(Val(oDCmAmount:TextValue))
     oDCCentsText:Value :=;
          CDecimal(Val(oDCmAmount:TextValue),2);
                + "/100"
   ENDIF
RETURN NIL
```

Note: The cWhole() and cDecimal() functions are located in the App:Misc module.

A single parameter, oCE, is passed automatically by CA-Visual Objects to the method. This is a ControlEvent object, which contains information about the event that just occurred.

The control event contains the control object, for which the event was generated, as an access method. The supplied code places the control object into a local variable in the declaration statement, as follows:

```
LOCAL oC := oCE:Control
```

You want to check for EditChange events which occur in the Amount field. The Amount field, on the NewPaymentWindow window, is named MAmount.

You can access a control's name using its NameSym variable, as follows:

```
oC:NameSym
```

Note: In this case, you only want to do something when the event is for the MAmount control, therefore, enclose your code inside an IF statement. For example:

```
IF oC:NameSym == #MAmount
   ...
ENDIF
```

If you have more than one control to handle, you can use a DO CASE statement in its place.

If the MAmount control is modified, the following code sets the values of the oDCDollarText and oDCCentsText fixed text controls (see the "Adding Controls to Your Windows" lesson in this guide for more information on controls):

```
oDCDollarsText:Value :=;
   CWhole(Val(oDCmAmount:TextValue))
oDCCentsText:Value :=;
   CDecimal(Val(oDCmAmount:TextValue),2) +;
         "/100"
```

The CWhole() function returns the word format version of the dollars portion (or whole part) of a numeric. The CDecimal() function returns the string representation (or decimal part) of the cents portion of a numeric.

Setting the oDCDollarsText and oDCCentsText fixed text controls to new values automatically repaints them on the window.

Notice the call:

```
SUPER:EditChange(oCE)
```

This is done because your customization is an enhancement to the current behavior of the control. You are not replacing it. However, many other things could be happening in the default behavior that, if unattended, could cause unpredictable results.

3. Save the source code by selecting the Save command from the File menu.
4. Close the Source Code Editor by double-clicking on its system menu.
5. Close the Window Editor by double-clicking on its system menu.

6. Build the application by selecting the Build toolbar button.

7. Run the application by selecting the Execute toolbar button.

Verify the results using the following steps:

1. Select the New command from the File menu.

   The New Record dialog box appears:

2. Select the Payment radio button and then choose OK.

   The New Payment window appears.

3. Type a dollar value into the single-line edit control designated for the amount paid.

   As you type the amount, it is printed across the payment receipt in word format.

4. Close the South Seas Adventures application by double-clicking on its system menu.

   As described earlier, the EditChange() method is called when any of the edit controls on the window are modified. Similarly, the ListBoxSelect() method is shared by list boxes and combo boxes while the ButtonClick() method is shared by radio buttons and push buttons.

Expose Event

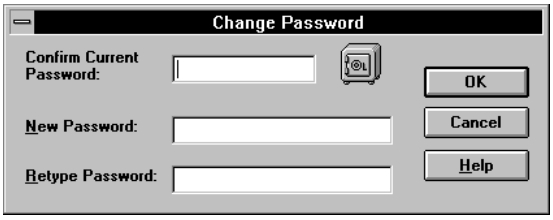
Expose events provide a method for allowing a window to be repainted.

The Expose() method is invoked when a window:

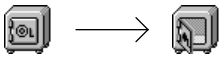
- Is first shown
- Is partially uncovered by another window
- Increases in size
- Is being restored after being iconized

In this part of the exercise, the DrawObject classes are used to demonstrate the use of the Expose() event handler method. The DrawObject classes provide a mechanism for painting a window with other displayable objects, such as bitmaps and ellipses.

In the South Seas Adventures application, the Change Password dialog box (displayed by choosing the Change Password command from the Options menu) allows the user who is currently logged in to modify their password.

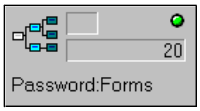


The user must first enter the current password and then type in a new password twice. If the current password is typed correctly, the safe, located next to the edit control, opens.



This was accomplished by combining an EditChange() event handler with an Expose() event handler.

1. Open the Password:Forms module by double-clicking on its module button.

   The _NewPasswordDialog window was created using the Window Editor. Its generated _NewPasswordDialog class, was then subclassed to the NewPasswordDialog class.

2. Find the NewPasswordDialog class and double-click on its entity button.

```
CLASS NewPasswordDialog INHERIT
_NewPasswordDialog
   PROTECT LockedBitmap
   PROTECT UnlockedBitmap
   PROTECT oCurrentBitmap
   PROTECT Server
```

   This allows the addition of the instance variables needed for this specialized window. These variables are assigned in the Init() method of the NewPasswordDialog class.

3. Find the NewPasswordDialog:Init() method and double-click on its entity button.

   The oCurrentBitmap variable initially contains the “locked safe” bitmap object. It is changed to the “unlocked safe” bitmap object in the NewPasswordDialog:EditChange() method, depending on the status of the Confirm Current Password edit control.

   Initially, it is set to the locked safe bitmap, since the edit control is blank. When the window is first displayed, the window Expose() method is invoked.

4. Find the NewPasswordDialog:Expose() method and double-click on its entity button.

```
METHOD Expose(oEE) CLASS NewPasswordDialog

   SUPER:Expose(oEE)

   // If the exposed area includes the bitmap
   // area
   IF oCurrentBitmap:BoundingBox:;
          Overlap(oEE:ExposedArea)
          // Draw the current bitmap
          SELF:Draw(oCurrentBitmap)
   ENDIF

   RETURN NIL
```

   This method first calls the method of the same name in the superclass to ensure that the normal refreshing takes place.

   Next, the customization of the Expose() method is added. To display a DrawObject object, the Window:Draw() method is used, passing it the object to be drawn as a parameter. However, the

safe bitmaps are only drawn if the newly exposed area covers the bitmap.

As discussed above, the Expose event is only triggered under certain circumstances. When the user enters the correct password, the bitmap must be updated on the window. You must force a repaint of the window.

5. Find the NewPasswordDialog:EditChange() method and double-click on its entity button.

   To force a repaint of the window, the Window:Repaint() and WindowRepaintBoundingBox() methods are used. These methods create Expose events on the window.

```
METHOD EditChange(oCE) CLASS NewPasswordDialog

    LOCAL oControl := oCE:Control

    SUPER:EditChange(oCE)

    ...

    // The edit control contains the current
    // password?
    IF Upper(;
           Trim(oDCCurrentPasswordSLE:TextValue));
                  == Upper(Trim(Server:Password))

           // Set the current bitmap to the
           // UNLOCKED safe.
           oCurrentBitmap := UNLOCKEDBITMAP

           // Move the focus to the next SLE
           oDCNewPasswordSLE:SetFocus()
    ELSE
           // Set the current bitmap to the LOCKED
           // safe.
           oCurrentBitmap := LOCKEDBITMAP
    ENDIF

           // Force a repaint of the bitmap area
    SELF:RepaintBoundingBox;
           (oCurrentBitmap:BoundingBox)
...

RETURN NIL
```

   When the user types into the Confirm Current Password edit control, its value is compared against the user's password in the database. If the two values match, oCurrentBitmap is updated to

the UnlockedBitmap object. If the two values do not match, oCurrentBitmap is updated to the LockedBitmap object. The SELF:RepaintBoundingBox() method is then invoked.

The SELF:RepaintBoundingBox() is used, since only this portion of the window is being modified. This type of optimization can be crucial when designing complex windows with many different displayed objects.

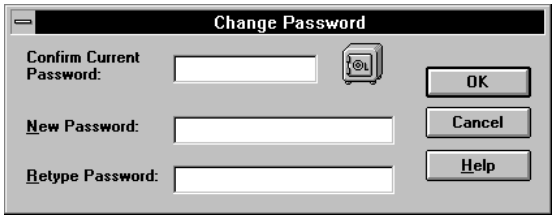
6. Close the Source Code Editor by double-clicking on its system menu.

7. Run the application by selecting the Execute button on the toolbar.

Verify the results using the following steps:

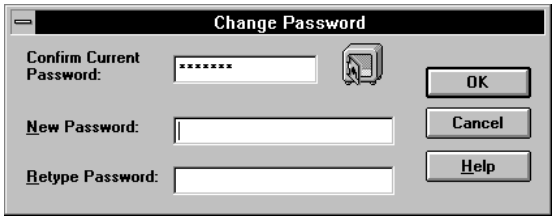
1. Log in to the application as **user**.
2. Enter **trainee** as the password.
3. Select the Change Password command from the Options menu.

   The Change Password dialog box appears:



4. In the Confirm Current Password edit control, type **trainee**.

   The safe now opens and the focus moves to the New Password control.



5. Choose Cancel to close the dialog box.
6. Close the South Seas Adventures application by double-clicking on its system menu.

Notify Events

The NotifyEvent event only occurs for data windows and is created by an attached server. This event is crucial for a data window, since the data window uses this handler to keep itself in sync with its attached server.

For example, if the oServer:Skip() method is invoked, the server first notifies the data window of its intention to change records. This gives the data window the opportunity to save any of the edit controls on its window to the current record before the record pointer is moved. Once control is returned to the server, it then moves the pointer. The server then notifies the data window that it has repositioned the record pointer. The data window then updates its controls from the server.

Your program can make use of this mechanism.

The example you are about to see uses the Notify() method to update the window caption. Each time the record pointer moves, the window caption reflects the new record.

1. Open the Item:Methods module by double-clicking on its Module Browser button.
2. Find the EditItemWindow:Notify() method and double-click on its entity button.

```
METHOD Notify(kNotifyName,uDescription);
          CLASS EditItemWindow
   LOCAL xRetNotify := ;
          SUPER:Notify(kNotifyName,uDescription)

   // Set new window caption
   IF kNotifyName >= NOTIFYFIELDCHANGE
          SELF:Caption := ;
                Trim(GetToken(SELF:Caption,1,;
                "-")) + " - " +;
                Trim(SELF:Server:Item_ID)
   ENDIF

   // If something has changed...
   IF kNotifyName == NOTIFYFIELDCHANGE
          SELF:Owner:BroadcastMessage(SELF,#Item)
   ENDIF

RETURN xRetNotify
```

The Notify() method receives a constant rather than an Event object. This constant identifies the event that occurred in the attached server. The possible values are prioritized and guaranteed to be in a specific order, thus you can use the following code,

```
IF kNotifyName >= NOTIFYFIELDCHANGE
```

to identify any event that involves a field change.

Notify() also receives a second parameter, <*uDescription*>, which is not always used.

For more information on the Notify() method, see the "DataWindow Class" topic in the CA-Visual Objects online Help.

When the Notify() method is invoked, the window captions are modified to reflect the current Item_ID.

```
SELF:Caption := ;
   Trim(GetToken(SELF:Caption, 1, "-")) +;
          " - " +;
          Trim(SELF:Server:Item_ID)
```

3. Run the application by selecting the Execute toolbar button.

Verify the results, using the following steps:

1. Select the Open command from the File menu.

   The Open File dialog appears:

2. Select the Item radio button and click OK.

3. When the Item browser appears, select an item from the browser and click on the Edit toolbar button.

   The Edit Item window appears:

4. To verify that the Notify() method is working, make changes and return to the Item browser; then click on the record movement buttons on the toolbar to view the changes.

5. Close the South Seas Adventures application by double-clicking on its system menu.

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

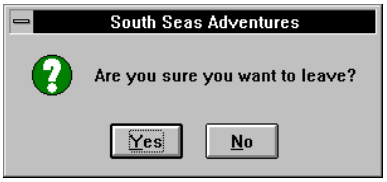
QueryClose Event

The QueryClose event occurs when a request has been made to close a window. This request can be brought about by double-clicking on a window's system menu or invoking a window's EndWindow() method, among other actions.

These two actions post a WM_Close message in the applications event queue. The CA-Visual Objects dispatcher then invokes the window's QueryClose() method.

The return value of your QueryClose() method determines if the window gets closed. If your QueryClose() method returns TRUE, the window is closed; otherwise, it remains open.

This can be very useful. The example you will examine uses the QueryClose() method to prompt the user prior to exiting the application, as in the following figure:



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

2. Select the Edit All Source in Module toolbar button.

3. Find the SSAWindow:FileExit() method and double-click on its entity button.

```
METHOD FileExit() CLASS SSAWindow

   SELF:EndWindow()

RETURN NIL
```

SSAWindow is the South Seas Adventures shell window. In an MDI application, double-clicking on the shell window's system menu is analogous to requesting to exit the application. You also want this behavior when the user selects the Exit command from the File menu. To provide this functionality, invoke the window's EndWindow() method.

4. Find the SSAWindow:QueryClose() method and double-click on its entity button.

```
METHOD QueryClose(oEvent) CLASS SSAWindow

   LOCAL oWB AS WarningBox
   LOCAL lLeave := FALSE AS LOGIC
   SUPER:QueryClose(oEvent)

   // Prompt the user before exit
   oWB := WarningBox{SELF,;
          "South Seas Adventures",;
          "Are you sure you want to leave?" }
   oWB:Type := BOXICONQUESTIONMARK + BUTTONYESNO

   IF oWB:Show() == BOXREPLYYES
          lLeave := TRUE
   ENDIF
```

```
...

RETURN lLeave
```

When the user chooses to exit the application, a warning box provides an opportunity to keep the application open.

```
// Prompt the user before exit
oWB := WarningBox{SELF,;
       "South Seas Adventures",;
       "Are you sure you want to leave?"}
oWB:Type := BOXICONQUESTIONMARK + BUTTONYESNO

IF oWB:Show() == BOXREPLYYES
```

If the user chooses Yes, the return parameter, lLeave is changed to TRUE and the window is allowed to close.

5. Find the SSAWindow:CloseAllChildren() method and double-click on its entity button.

```
METHOD CloseAllChildren() CLASS SSAWindow

LOCAL i AS WORD
LOCAL wLen AS WORD
LOCAL lSuccess AS LOGIC
LOCAL aTmpChildWindows
lSuccess := TRUE

aTmpChildWindows := AClone(aChildWindows)
wLen := ALen(aTmpChildWindows)

FOR i := 1 TO wLen
       aTmpChildWindows[i]:EndWindow()
NEXT

RETURN (Len(aChildWindows) == 0)
```

If all of the child windows end up closing, the CloseAllChildren() method returns TRUE. The QueryClose() method also returns TRUE, thereby freeing the way for CA-Visual Objects to continue closing the application.

Summary

In this lesson, you have learned about events and event-driven applications. You now know how CA-Visual Objects receives and dispatches events. Additionally, you have seen how to customize your own windows, using a wide variety of event handlers.

The next lesson shows you how to use icons and cursors to enhance the “look” of your applications.