The App:Start() Method

Every application you create in CA-Visual Objects requires some function or method named Start() that serves to get the application started; in the Standard Program, this is the App:Start() method. (You were prompted for the name of the module to contain this code in the Generate Application Framework dialog box presented earlier in this lesson.)

Since we opened up App:Start() in the last section, you should still be viewing the source code for this method in the Source Code Editor. It looks something like this:

```
METHOD Start() CLASS App
    LOCAL oWindow AS Window
    Enable3DControls()
    oWindow := StandardShellWindow{SELF}
    oWindow:Show()
    SELF:Exec()
```

What this does is declare, create, and show the shell window (it's called *oWindow* and is instantiated using the StandardShellWindow class described in the next section). It also enables a sculpted, 3-D look in your application, and then calls the App:Exec() method—actually, the source code reads "SELF:Exec()," but that is easily explained.

Before the App:Start() method is invoked, the App object is created by the system automatically (that's why we called the App object "invisible" earlier). You never need to instantiate an object of the App class (as you do with all other objects used in an application), just as you never need to explicitly invoke the App:Start() method. The system does all of this for you in order to get your application started.

Once the App object exists and its Start() method is executing, App calls its own methods—for example, Exec()—by referring to them using SELF. SELF is a special keyword that refers to the object whose method is currently executing—for example, SELF:Exec(). You also see the SELF keyword used to instantiate the StandardShellWindow class, which is further explained in the next section.

Note: Within the App:Start() method you ***must*** call App:Exec(). Doing so invokes the Windows event handling loop for your application, as well as the CA-Visual Objects event handling mechanism. Unless you invoke App:Exec(), the system cannot start sending you events. (See Default Event and Error Handling below.)

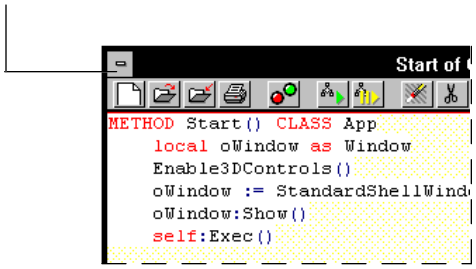
The Shell Window

Now that you have seen the App:Start() method, you understand how the application gets started, so let's take a look at the shell window for this application. It is defined in the Standard Shell module.

Assuming you are still viewing the App:Start() source code:

1. Double-click on the Source Code Editor's system menu to close it:

   Double-click to close this window

   You are returned to the application-wide Entity Browser.

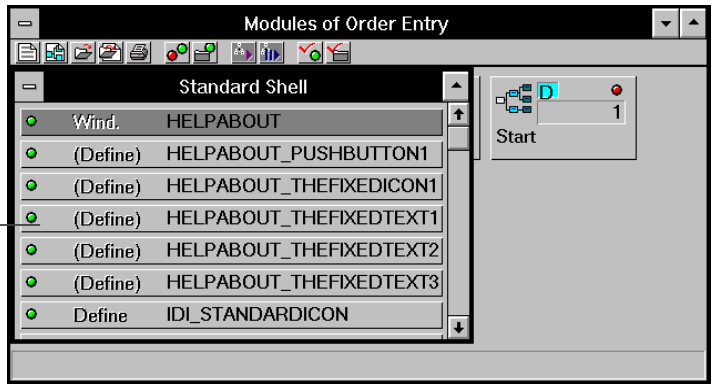
2. Double-click on the Entity Browser's system menu to close it.

   You are returned to the Module Browser.

3. Double-click on the Standard Shell module to open an Entity Browser for it.

This opens a module-wide Entity Browser for the Standard Shell module:

Module-wide Entity Browser for Standard Shell module



Recall that this type of Entity Browser displays only the entities in a particular module. In this section, we will be taking a closer look at the Standard Shell module as a whole; therefore, this Entity Browser will be more convenient to use than the application-wide Entity Browser.

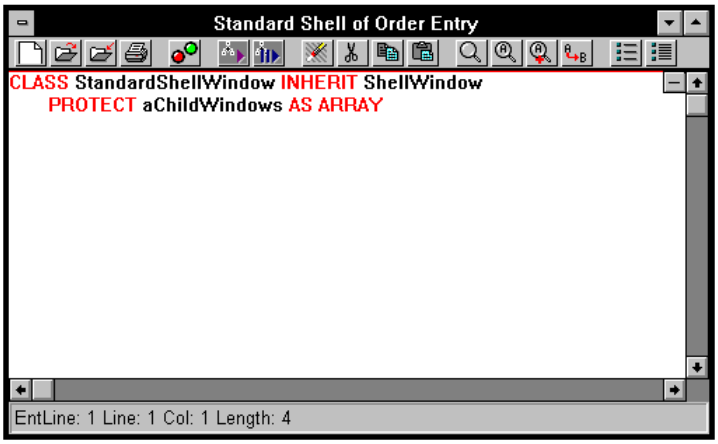
StandardShellWindow

The shell window for this MDI application was *instantiated* in the App:Start() method using the StandardShellWindow class:

```
oWindow := StandardShellWindow{SELF}
```

You will see the class *declaration* for StandardShellWindow if you scroll this Entity Browser down once or twice (i.e., Class STANDARDSHELLWINDOW).

Double-click on it to view the source code, and you will see that this class derives from the ShellWindow class:



Note: The second line of code defines an array that will be used to track the number of child windows opened in the application.

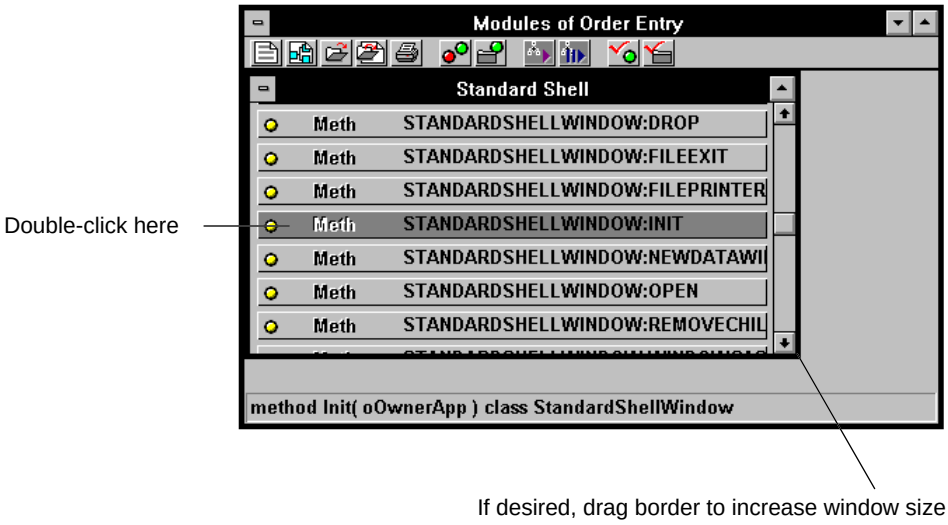
The ShellWindow class resides in the GUI Classes library—as mentioned earlier, this class is already configured to support MDI. Since StandardShellWindow inherits from ShellWindow, it is preconfigured for MDI support as well.

Init() Method

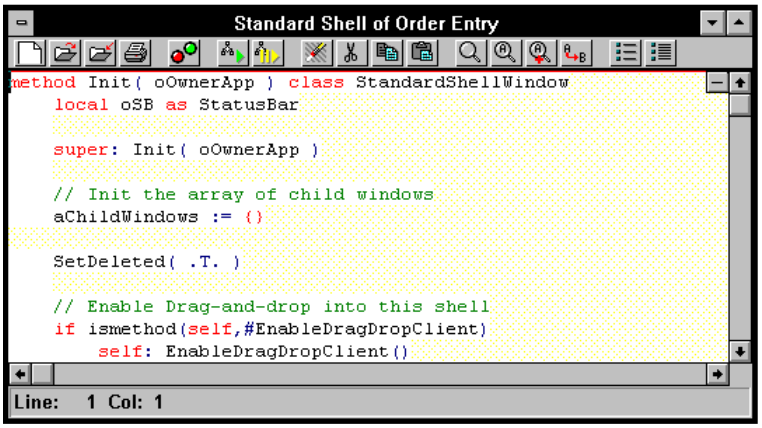
Close the Source Code Editor and when you return to the Entity Browser, scroll through the list—you will see several methods defined for the StandardShellWindow class.

Among the most important methods is StandardShellWindow's Init() method. It is called automatically by App:Start() upon instantiation of the StandardShellWindow class.

Scroll down to the StandardShellWindow:Init() method and double-click on it to view it in the Source Code Editor, so that we can see what it does and do a little customizing.



After double-clicking on the Init() method, it is loaded in the Source Code Editor:



The first thing you notice is that this method takes a parameter called *oOwnerApp*. If you remember, the App:Start() method instantiated this class using SELF as an argument, so the window knows who its owner is (the App object).

Then, StandardShellWindow:Init() invokes the Init() method of its superclass, ShellWindow. This has two effects:

- It ensures that the internal data of the ShellWindow class is properly initialized—that is the job of any Init() method.
- It registers the App object itself as the *owner* of the StandardShellWindow.

As you scroll through the rest of the method, you can see that there is some code to initialize the array that will be used to track the number of child windows opened in the application; to enable the window as a drag-and-drop client; to enable a status bar and put some information on it; to attach a menu to the window (more on this below); and to define an icon for the window when it is minimized. (This icon is also defined in the Standard Shell module as the IDI_STANDARDICON resource and constant.)

Finally, there is code to create a caption (or title) for the window, which you will now customize for this application. To do this:

1. Move the cursor to the line of code reading:

   ```
   SELF:Caption := "Standard MDI Application"
   ```

2. Change it to:

   ```
   SELF:Caption := "Order Entry"
   ```

Close the Source Code Editor (just double-click on its system menu) and save your change by choosing Yes when prompted.

Other Methods

When you return to the Entity Browser, you can see that beyond the Init() method, StandardShellWindow has methods to respond to events generated by its menu, including one called Close() to shut down the application (by calling the App:Quit() method) when the user closes the shell window.

When you are finished browsing through the methods, you can close the Entity Browser by double-clicking on its system menu.

The Empty Shell Window

When no child windows are open in the shell window, it is referred to as the *empty* shell window. This is the state of the shell window when

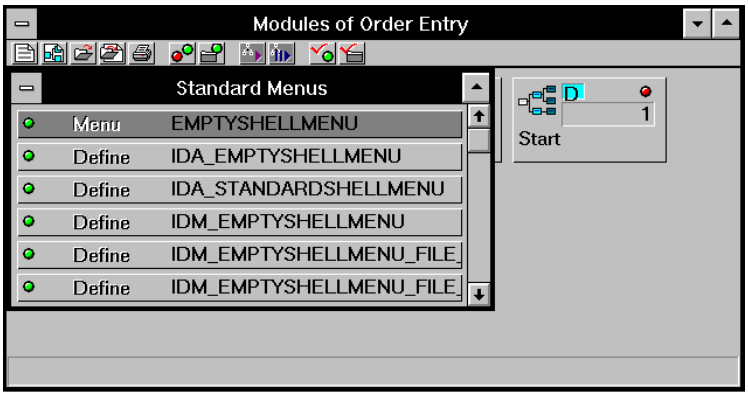
it is initially created by App:Start(), and the menu associated with the empty shell window is necessarily sparse because of the limited actions that you can perform when no data is present to manipulate.

EmptyShellMenu

In StandardShellWindow:Init(), the menu attached to the window was instantiated with the following line of code:

```
SELF:Menu := EmptyShellMenu{SELF}
```

This EmptyShellMenu class is defined in the Standard Menus module. Access the Entity Browser for this module and scroll through it:



Along with the EmptyShellMenu class, you will see an EmptyShellMenu_Accelerator class, an EmptyShellMenu menu entity, resource entities for the menu and the accelerator, several defines to identify properties of the menu, and Init() methods to instantiate both classes mentioned. All of this code was generated by the Menu Editor, which you will look at next.

Resource Entities

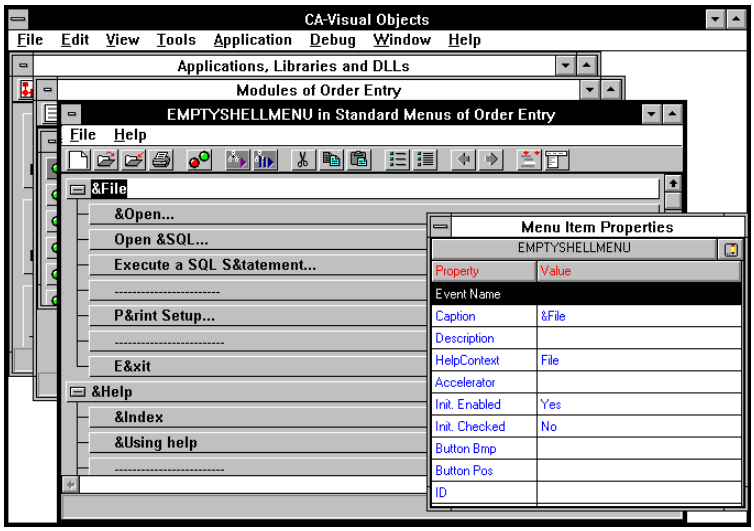
For example, to define the empty shell menu and its accelerator keys in a way that is understood by Windows, there are two resource entities, RCMenu IDM_EMPTYSHELLMENU and RCAccl IDA_EMPTYSHELLMENU. You can look at these if you like; they contain source code in the format that the Windows resource compiler wants.

The Menu Entity

The actual menu entity contains information that allows the Menu Editor to graphically display the menu layout so that you can edit it. For example, EmptyShellMenu contains File and Help menus that are

used to open child windows, set up printers, exit the application, and get help.

If you like, you can double-click on the EmptyShellMenu menu entity to get an idea of what this menu looks like in the Menu Editor (you'll actually learn how to use this editor later in this chapter).



As you can see, EmptyShellMenu has just two menus: File and Help. Note that the Menu Editor allows you to design a toolbar to be associated with a menu. As part of the Standard Program, EmptyShellMenu is paired with a toolbar—when you later attach EmptyShellMenu to a window, the toolbar is automatically pulled in at the same time.

Note: The toolbar is not part of the display that you normally see in the Menu Editor. You can view it using the File Preview Toolbar command.

When you are through looking at the menu, close the Menu Editor by double-clicking on its system menu.