

Visual Objects

For
Windows 2000® and Windows XP®

South Seas Adventures
An Exploration of VisualObjects
Version 2.7



Contents

Chapter 1: Introduction

Welcome to South Seas Adventures!	13
Using the Integrated Development Environment	15
Creating the Application Building Blocks	15
What You Should Know	16
What Are Objects?	16
What Can Objects Do?	17
Are Objects Defined?	17
How Are Objects Created and Destroyed?	17
How Do Objects Interact?	17
When Does It All Begin and End?	18
How Are Objects Used in South Seas Adventures?	18
South Seas Adventures Application Design	19
Creating the Primary Building Blocks	19
Data Tables, Servers, and Fields	19
Data Windows	20
Shell and Dialog Windows	21
Window Controls	21
Event Handlers	22
Reports	22
Help Systems	23
Icons, Cursors, and Draw Objects	23
Linking the Primary Building Blocks	24
Completing the Remaining Building Blocks	24
Developer-Coded Entities	25
System-Generated Entities	26
Linking the Remaining Building Blocks	26
Begin Your South Seas Adventure... ..	27

Chapter 2: Exploring the CA-Visual Objects 2.7 Integrated Development Environment

Overview	29
Choosing a Directory Structure	29
The South Seas Directory Structure	30
Creating Path-Independent Applications	31
The CA-Visual Objects 2.7 Multi-Tiered Repository	31
Application Component Hierarchy	32
Automated Make and Entity-Level Compiling	32
Grouping Your Entities into Modules	32
Module Design Considerations	32
Module Naming Conventions	33
Exercise	35
Using the Repository Explorer	35
Importing the Application	37
Configuring Your Application Environment	38
Application Options	38
Compiler Options	39
Viewing the Application Modules	40
Building the Application	41
Running the Application	42
Running the Program Dynamically	42
Creating and Running an Executable File	43
Summary	43

Chapter 3: Working with Data Servers

Overview	45
Exercise	47
Creating a Customer Data Server	47
Invoking the DB Server Editor	47
Importing a .DBF File	49
Importing an Index	51
Saving the Data Server	53
Creating a SQL Server	53
Installing ODBC Drivers	54
The ODBC Administrator	55
Using the SQL Editor	58

Programming with Servers	60
Importing a Support Module	60
Viewing the Server Source Code	61
Running the Application	62
Event Notification	62
Client Data Forms	62
Child Servers	63
Manual Notification	64
Broadcast Message Activation	65
Summary	67

Chapter 4: Defining Field Specifications

Overview	69
Exercise	70
Creating and Modifying Field Specifications	70
Planning Data Server Field Properties	71
Attaching a Field Spec to a Data Server Field	72
Creating Field Specs from the DB Server Editor	74
Summary	75

Chapter 5: Creating and Using Windows

Overview	77
Single Document Interface Applications	77
Top Application Windows	77
Multiple Document Interface Applications	78
Shell Forms	78
Dialog Forms	78
DataDialog Forms	79
Child Application Windows	79
Data Forms	80
Server Use	80
Data Propagation	80
Form and Browse View	80
Data Validation	81
Using the Window Editor	81
Disconnected Controls	81

Exercise	82
Viewing a MDI Application	82
The Shell Form	82
Adding Functionality	82
Creating a Modal Dialog Box	84
Warning Box Modal Dialog Forms	84
Retrieving Values from Modal Dialog Forms	85
Creating a Data Form	87
Importing a Support Module	88
Creating a Data Window Template	88
Designing Your Window Layout	90
Adding a Push Button	91
Compiling and Testing Your Changes	93
Summary	94

Chapter 6: Adding Controls to Your Windows

Overview	95
Exercise	96
Single-line Edit (SLE) Controls	96
Multiline Edit (MLE) Controls	98
Moving the MLE Control	98
Viewing Your Results	98
Combo Box Controls	99
Check Box Controls	102
Radio Button and Radio Button Group Controls	102
List Box Controls	105
Group Box Controls	107
Fixed Icon Controls	108
Push Button Controls	109
Programming Techniques	110
Tab and Group Stops	110
Control Order and Multiple Groups	112
Naming Controls	112
Summary	113

Chapter 7: Inheritance and Subclassing

Overview	115
Exercise	117
Customizing Generated Code	117
Summary	118

Chapter 8: Creating Menus and Toolbars

Overview	119
Exercise	119
Creating a New Module	119
Creating the Menu	120
Using Auto Layout	121
Previewing Your Menu	122
Collapsing/Expanding the Menu Structure	123
Adding an Item to the Hierarchy	123
Changing the Hierarchy of a Menu Item	124
Removing Menu Items from the Hierarchy	125
Specifying Menu Actions to Perform	125
SSAWindow Event Name	126
Providing Menu Shortcuts	126
Checking a Menu Item	127
Creating a Toolbar	128
Changing Toolbar Button Positions	130
Spacing Between Toolbar Buttons	131
Other Modifications to the Toolbar	132
Saving the Menu	132
Attaching a Menu to a Data Window	133
Putting It All Together	134
Designing a Menu	135
Customizing a Menu	135
Disabling Menu Items	135
Editing Toolbar Buttons	136
Summary	138

Chapter 9: Accessing and Updating Data

Overview	139
Narrative	139
Xbase Compatibility	139
Access and Assign Methods	140
Generated Data Server Classes	141
Base DBServer and SQL Classes	142
Data Forms	143
Data Servers Attached to Data Forms	144
Controls	144
Summary	146

Chapter 10: Customizing Window Event Handlers

Overview	147
Exercise	149
Using the EditChange() Method	150
Viewing Your Results	152
Using the Notify() Event Handler	153
Creating the Method	153
Viewing Your Results	154
Using the QueryClose() Event Handler	155
Summary	156

Chapter 11: Working with Icons and Cursors

Overview	157
Creating an Icon	157
Saving the Icon	160
Attaching Icons to Data Forms	161
Labeling Your Application with an Icon	162
Icons in the Program Group	162
Attaching Icons to Shell Forms	164
Displaying an Icon on a Window	165
Using Predefined Cursors	166
Creating and Modifying Cursors	166
Summary	169

Chapter 12: Working with Draw Objects

Overview	171
Exercise	172
Making the Dialog Box Resizable	173
The Resize Event	174
Using Bitmaps	175
Declaring a .BMP File as a Resource	175
Creating a Bitmap Object	176
Drawing a Bitmap on a Window	177
Using Text Objects	178
Dynamic Positioning of Controls	179
Viewing the Results in the Application	180
Summary	180

Chapter 13: Reporting with the Report Editor

Exercise	181
Using the Report Editor	181
A Quick Tour	187
Adding Your Personal Touch	188
Saving Your Work	191
Running Your Report Within Your Application	191
Report Parameters	192
Passing Parameters to the Report Editor from CA-Visual Objects	195
Verifying the Results	197
Summary	198

Chapter 14: Debugging Your Application

Error Browser Exercise	199
Importing a Module with Errors	200
Resolving the Errors	201
Debugger Exercise	203
Viewing the Error	204
Introducing an Error	205
Set Debugging On	206
Set Debugging at the Module Level	206
Running the Application Using the Debugger	207
Locating the Bug	208
Evaluating Expressions	209
Correcting the Error	210
Summary	211

Chapter 15: Adding Help to Your Applications

Overview	213
Context-Sensitive Help	213
Exercise	214
Implementing Context-Sensitive Help	214
Attaching Your Help File	214
Assigning Help to a Window	216
Assigning Help to a Control	217
Assigning Help to a Menu Command	217
Invoking Context-Sensitive Help	218
Viewing Help for a Control	219
Viewing Help for a Window	219
Viewing Help for a Menu Command	220
Implementing Direct Calls to Help	221
Menu Commands	221
Creating Help Files	222
Topic Files	222
Project File	222
Summary	223

Chapter 16: Using Win32 API Functions

Overview	225
Exercise	226
Windows Metric Information	226
Summary	227

Chapter 17: Using Libraries and Dynamic Link Libraries

Overview	229
Libraries	229
Dynamic Link Libraries	229
Exercise	230
Creating and Using a Library	230
Creating a New Library Application	231
Moving Modules Between Applications	232
Building the Library	233
Using the Library	233
Creating and Using a DLL	234
Creating a New DLL Application	234
Copying Modules Between Applications	234
Using a DLL	235
Creating a .DLL File	236
Using a CA-Visual Objects .DLL	236
Summary	238

Chapter 18: Distributing Your Application

Overview	239
Exercise	240
Generating the Executable	240
Using the Install Maker	240
Specifying Other Files for Installation	242
Specifying File Properties	243
Creating a Project File	244
Creating Disk Images	245
Creating and Testing the Distribution Disks	245
Summary	246

Appendix A: Creating a Path-Independent Application

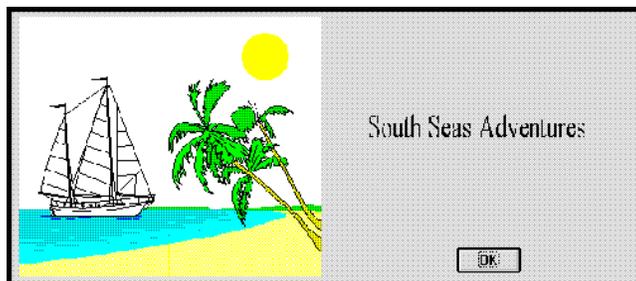
Establishing Drive and Directory Independence.....	248
Help Files	249
DB Server Data Files	249
Report Files	250
Icon, Cursor, and Bitmap Files	251
Summary	252

Index

Introduction

Welcome to South Seas Adventures!

Take a hands-on tour of a sample application, South Seas Adventures, and discover how quickly and easily you can begin developing your own applications in CA-Visual Objects 2.7. This application has been partially developed—allowing you to complete the development process as you work through the tutorial:



South Seas Adventures simulates an operation support system that might be used by the employees of a hypothetical company, South Seas, Inc. Employees of the company act as booking agents to help customers plan an “adventure,” consisting of several vacation activities—such as parasailing, jet skiing, or dinner cruises.

Additionally, South Seas Adventures manages many other aspects of the business, including subsystems to handle customer, employee, order-entry, invoice, and payment information.

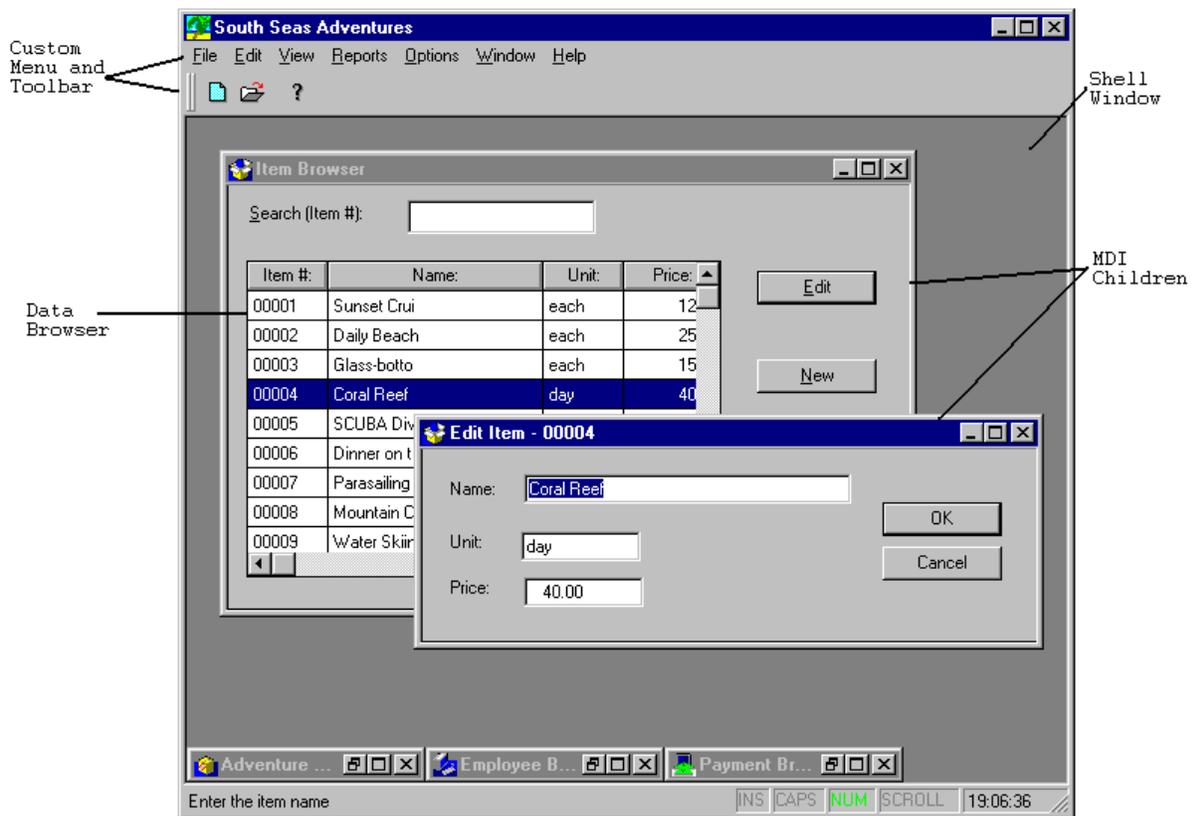
Some of the system operations that are covered in the tutorial are listed below:

- Creation of an order for an adventure
- Creation of an invoice
- Recording of payment information
- Preparing customer reports
- Exporting data to an external SQL-based accounting system

Not only does the application have all the proper components to satisfy the needs of South Seas, Inc. operations, it also employs many user-friendly features, such as windows and toolbars, found in your favorite Windows applications.

South Seas Adventures is an MDI (multiple document interface) application that features a consistent, easy-to-use interface that demonstrates how standard Windows features are created, using the CA-Visual Objects 2.7 IDE (integrated development environment). The application allows you to view or browse through information easily, as well as create or delete a customer record.

Some of these standard Windows features are shown below:



You will now discover how the application is developed—using the CA-Visual Objects 2.7 IDE tools.

Using the Integrated Development Environment

CA-Visual Objects provides both a visual and incremental development environment. It integrates the many different types of building blocks, or entities, required to build a Windows application like South Seas Adventures. Visual editors allow you to create windows, menus, and other objects. Even more importantly, they automatically generate the related source code entities. You can then create and edit the relatively small number of source code entities that define what happens when a push button is pressed or a menu command is selected. Finally, you can create source code entities that incorporate business and detailed program logic rules.

CA-Visual Objects 2.7
Multi-Tiered
Repository

All of the entities are stored in the CA-Visual Objects multi-tiered repository. The repository allows you access to all of the entities associated with an application. The entities are organized into modules so that similar entities can be grouped together to make them easy to locate.

The CA-Visual Objects IDE is well-suited for incremental development. You can create the data servers, windows, and menus in any order. Once the interface is completed, you can start adding source code entities that define what happens when a push button or menu command is selected. The repository manages all the building blocks, compiles the changed entities, and allows you to run the application, so that you can revise the overall design and recreate the building blocks in an incremental fashion.

Creating the Application Building Blocks

You will have the opportunity to add some of the key building blocks to the South Seas Adventures application. Since certain exercises require the use of a building block created in an earlier chapter, you should work through the chapters in sequential order.

After you have completed the development of the application, it will be capable of accessing and updating information about customers, employees, adventures, activities, invoices, and payments. This is accomplished through the use of a variety data-aware windows, dialog windows, menus, and reports.

Here are the building blocks that you will create:

- A data server for customer data, based on the Xbase model of a .DBF file, and two SQL servers for data to be exported to an accounting system
- General field specifications, or *field specs*, that can be used with existing data servers
- A data-aware window to edit customer data, using the Auto Layout feature with a data server
- Controls for the customer data window
- A menu and toolbar for the customer data window
- An icon for the customer edit window
- A resizable opening dialog box with a bitmap and a text object
- A report that lists customers
- A library and DLL for .INI file management activities

What You Should Know

The South Seas Adventures application was created using the CA-Visual Objects object-oriented development system. If you are not familiar with object-oriented systems, you can use the following background information to assist you in getting the most out of this tutorial.

What Are Objects?

Objects are special types of programming entities—like windows, controls, menus, or toolbars—that have properties associated with them (like a border or caption) and can perform specific actions (like displaying itself on the screen, in the case of a window, or toggling the check mark indicator, in the case of a check box control).

These objects are generally created, and ultimately destroyed, in response to something the end user does via the application interface. Every time a user accesses a data table and it is connected to the application, a data server and related field spec objects are created. Also, special objects that relate to a problem-specific business process, such as creating an invoice, may be created.

What Can Objects Do?

They can walk, talk, and listen! The developer can specify what actions are possible for each type of object; this is done by creating source code entities called *methods*. An object can also provide information for use by other objects or by the business logic routines in the application; the way an object talks is defined in *access* source code entities. An object can also receive information via *assign* source code entities, which can modify its current state.

Are Objects Defined?

Objects are defined by a special kind of source code entity called a *class*. Approximately 90 percent of all classes you need are defined automatically by CA-Visual Objects. A class specifies the blueprint for each type of object and typically inherits most of its properties from another class, called the *parent* class.

However, in some situations you may want to define a class from scratch. A special `Init()` method defines all the related details of what must be done when the object is created. Additional methods can be defined to specify the other desired behaviors of the object, including what information can be received by it (assigns) or provided by it (accesses).

How Are Objects Created and Destroyed?

Objects are created in the runtime environment by the CA-Visual Objects 2.7 runtime system. End-user actions trigger source code that creates a new object; this process is called *instantiation* since a new instance of the object is created. Each object is given an internal name by the runtime system to allow it to manage the activities and communications for all the existing objects. Objects are destroyed as a result of end-user actions. For example, the menu, data server, and controls for a data window are destroyed when the end user closes a window. All windows are destroyed when the end user closes the application.

How Do Objects Interact?

All actions and communications between objects and other variables are managed by the CA-Visual Objects runtime system. When an end user starts an application, a well-defined environment, or universe, is created. Once objects are created in response to end-user actions, all the behaviors and information transfers progress according to the source code specifics. Each object is a well-behaved automaton that does what is asked of it, speaks only when spoken to, and accepts only certain kinds of information. An end user is the invisible hand that triggers the creation, activities, and destruction of each object.

When Does It All Begin and End?

The runtime universe begins when the user starts the application. This triggers the App:Start() method. The App:Start() method defines the shell window to be opened, and then triggers the running of the application with the App:Exec() method. The end user then defines all the other events, including the creation and destruction of objects. Finally, the application is closed when the App:Quit() method is activated.

How Are Objects Used in South Seas Adventures?

If you have not used objects before, you should be reassured by the fact that approximately 90 percent of all the classes, methods, accesses, and assigns in the South Seas Adventures application are generated automatically by CA-Visual Objects. Every time you use one of the visual editors and save whatever it is you are creating, all the related source code entities are generated.

This tutorial also highlights the few classes that are not generated automatically. These classes include dialog window classes that inherit properties of dialog windows generated by the Window Editor, an Invoice class created from scratch, and two file specification classes.

Creating Objects

When you create an object, it is named in the source code like any other variable, but the letter “o” is used as a prefix for the name. For example, since “oDC” indicates that an object is a data-aware control, a single-line edit control on a window could be named oDCFfirst_Name. It would then be instantiated with the following source code:

```
oDCFfirstName := SingleLineEdit{...}
```

This control is instantiated from the parent class (the standard SingleLineEdit class) according to the details and parameters within the {} braces. These curly braces indicate that an object is being instantiated.

Working with Objects

You can access character information from the object, by using the Value access of the parent SingleLineEdit class:

```
cTempName := oDCFfirstName:Value
```

Or, you can assign character information to the object with the Value assign of the parent class:

```
oDCFfirstName:Value := cNewName
```

If you want to clear the contents of this control, use the name of the object followed by the Clear() method defined for the parent class, as follows:

```
oDCFfirstName:Clear()
```

Finally, remember that all the interactions between objects are handled automatically by the CA-Visual Objects runtime engine. Once the end user starts the application, all these details should be of no concern to the developer.

That is all you really need to know about objects to benefit from the South Seas Adventures tutorial. Remember that as you create data servers, windows, and menus with the visual editors, the related object-oriented code is generated automatically. That means that the majority of your time as a developer can be spent designing the interface and business logic, while you leave the intricacies of creating a Windows application to CA-Visual Objects.

South Seas Adventures Application Design

Let's now begin to take a closer look at the internal workings of the South Seas Adventures application by examining its various components. This section can be used as a road map to what follows in the tutorial.

Creating the Primary Building Blocks

Entities that are created by the CA-Visual Objects visual editors are discussed in this section. You will learn how easily they can be created and incorporated into an application.

Data Tables, Servers, and Fields

The South Seas Adventures application provides access to information about customers, employees, adventures, activity items, invoices, and payments—the six primary types of business data used in the application—by means of the following data structures.

Data Tables

The types of tables that are needed to store primary and other types of data are listed below:

- Individual DBF tables for storing customer, employee, item, and payment information.
- Paired DBF tables for adventures and invoices: one for header information and the other for a varying number of detail records.

For example, an adventure is defined for a customer whose vacation travel spans a fixed period of time (header information) and contains one or more adventure activity items (detail information).

- SQL tables for accounting invoices (AccInv) and accounting payments (AccPay), simulating the external accounting system.
- DBF lookup tables (State, SysKey, and Tender).

Data Servers A data server must be created as an interface to each of these tables. You can use the DB Server and SQL Editors to accomplish this task quickly and easily. The IDE automatically creates the server entities and all the related field spec entities when you save a server in its editor.

The South Seas Adventures application requires data servers for the DBF tables and SQL data servers for the accounting tables. You will see how the Customer, AccInvc, and AccPay servers are created in the “Working with Data Servers” chapter.

Data Fields When a data server is created, the data fields for each server are defined in either the DB Server or SQL Editor. Typically, the server is created by importing a STRUCTURE from a .DBF file or an SQL table. If this is not done, the name, length, and type must be specified for each data field. The related field spec entities for each data field are automatically created when you save the server.

You will get the opportunity to examine the field specifications for various data fields within the South Seas Adventures application.

Data Windows

Data windows not only provide access to information that you need, but also allow the user to create, edit, and delete records. All the related source code entities (classes, methods, accesses, assigns, and defines) needed to perform these functions are created automatically when you save your window design.

The following data windows are used in the South Seas Adventures application:

- *Browser*—Contains a subform that displays a table showing several data fields for all records and an edit control for searching the table. One table is used for each business data type.
- *Subform*—Contains a multicolumn tabular display for use in the browser window. One table is used for each business data type.
- *New*—Used to create a new record. It applies to all business data types, except invoices (since invoices are created from the Edit Adventure window).
- *View*—Used for viewing payment information, which is not editable.
- *Edit*—Used to edit all fields of a single record. It applies to the customer, employee, and item business data types.
- *Master-Detail Edit*—Used to display header information and a detail subform. It applies to adventure and invoice business data types.
- *Detail Subform*—Contains a tabular display for use in a master-detail edit window. It applies to adventure and invoice detail data.

You will learn how to create the Edit Customer data window as part of this tutorial.

Shell and Dialog Windows

A shell window provides the framework for all the other windows and menus in your application. Dialog windows provide a way to communicate with the end user. These windows are also designed using the Window Editor; all the related source code entities are created automatically when you save the window design.

The South Seas Adventures application contains a single shell window and several dialog windows, including:

- Shell window: SSAWindow
- Navigational dialog windows: Opening, FileNew, FileOpen
- Print dialog windows: CustAdv, CustRpt, InvcRpt, PayRpt, Printer, PrintReport
- Other dialog windows: About, Find, Login, NewPassword, Progress

Window Controls

CA-Visual Objects has many types of controls that can be used in a window design. These controls are derived from standard classes. When you save a window design, the required source code for each control is automatically generated. Alternatively, you can make use of these controls by writing source code directly.

The majority of the controls are called “data-aware” controls, because they can be directly linked to a field in a data server, and therefore, can directly communicate with a field in a data table. When the Window Editor generates source code, the object name is prefixed with the three letters “oDC”.

A few types of controls are not data-aware—for instance, push buttons, radio buttons, fixed text objects, fixed icons, and group boxes. Object names for these controls are prefixed with “oCC.”

[Chapter 6: Adding Controls to Your Windows](#) steps you through the process of placing controls on the Edit Customer window created in [Chapter 5: Creating and Using Windows](#).

Menus allow a user to communicate with an application to perform an action. The Auto Layout feature of the Menu Editor can be used to get a quick start on your menu design. You can then add or delete menu items. You can also define a toolbar for the menu. When you save a menu design, the Menu Editor automatically generates all the source code entities for the menu.

The SSAShellMenu menu is the one associated with the main shell window that you see when the application begins. It allows you to select the next window to be opened. This *child* window has an associated SSACChildMenu menu that invokes various data windows.

In [Chapter 8: Creating Menus and Toolbars](#), you will create the CustomerMenu menu, which will then be attached to the Edit Customer window.

Event Handlers

Once the primary building blocks are created, you can begin to write the source code that defines what should happen, for instance, when the user activates a push button or a particular menu command. A Window class method (or event handler) can be created that holds the code that is executed when a push button is clicked. The Window Editor allows you to write this source code while designing a window, by invoking the Source Code Editor. In the Menu Editor, you must specify the name of a window, report, or method for each menu item.

In addition, you can create special methods that are activated when certain events occur in a window, for example, when any button on a window is clicked or the contents of any edit control are modified. This type of source code must be included in special window event handler methods, which can also be done easily while still in the Window Editor.

You will learn about window event handlers for the South Seas Adventures application in [Chapter 10: Customizing Window Event Handlers](#).

As you define the event handlers for your controls and menu items, the building blocks of your application are linked together. At this point in the development process, you can begin to run the application from the IDE to see how the various components work together.

Reports

Reporting can be a powerful feature in a business application. The CA-Visual Objects Report Editor allows you to specify which tables are to be used in a report by defining the data server names. Once you select a report style—tabular, form, labels, letter, or freestyle—a basic report definition is generated which you can then modify. When you save the report definition, an external REPORT EDITOR file (.RET) is generated, as well as several related source code entities.

There are several reports that are supplied with the South Seas Adventures application that you can examine. In [Chapter 13: Reporting with the Report Editor](#), you will also create a report that lists all customers.

Help Systems

Another primary building block of an application is the help system. This can be an important part of making your application easy to use.

Context-sensitive help can easily be integrated into your application. First, you must create a help file that has all the required help topics. You must then attach the help file to the application by specifying the help file name as a property of the shell window. An individual help topic can be designated in the visual editors for each window, control, menu item, server, and field. Thus, there are many linkages between the application and the help file.

You will add help to the South Seas Adventures application in [Chapter 15: Adding Help to Your Applications](#).

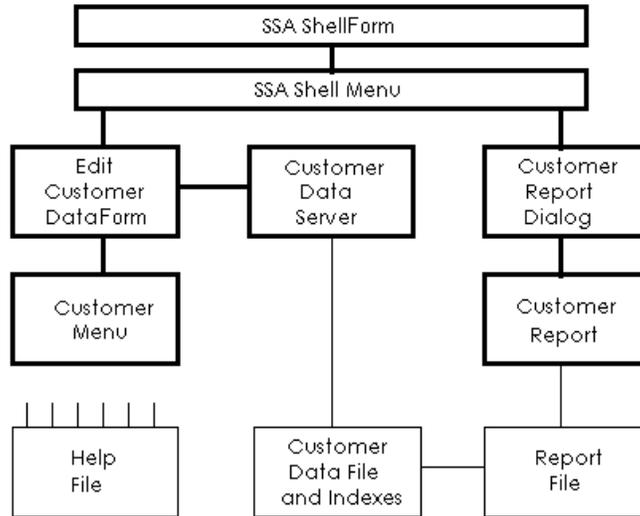
Icons, Cursors, and Draw Objects

In addition to the primary building blocks, there are a few more entities that you can add to your application that can enhance the “look” of your application, including icons and cursors.

You will create an icon (MyIcon) in [Chapter 11: Working with Icons and Cursors](#) of this guide. You will also see how to work with bitmaps and text objects in [Chapter 12: Working with Draw Objects](#) and learn how the Window:Draw() method can be used to display them.

Linking the Primary Building Blocks

The following diagram illustrates the basic structure of the South Seas Adventures application and the relationships among the primary building blocks (those that can be created using the editors in the CA-Visual Objects 2.7 IDE):



Note: Rectangles with thick borders are the primary building blocks, while those with thin borders are external files. Design linkages are shown as thick lines, while external linkages are shown with thin lines.

The main application window is the shell menu, which allows you to open a data or dialog window. Menus are associated with windows, and data servers act as the interface between data windows and external data files. There can be many linkages to the help file. Finally, each report is linked to an external report definition file which provides a linkage to the data files.

Completing the Remaining Building Blocks

At this point, you have learned how the CA-Visual Objects 2.7 visual editors help simplify the development process when creating many of the required application entities. Now, let's look at some of the remaining building blocks that you will need.

The following table summarizes how the different types of application entities are created:

Visual Editor Design Entities	Developer-Coded Entities	System-Generated Entities
Cursors	Functions	Defines
Data servers	Global variables	Resources
Field specs	Classes	Classes
Icons	Accesses	Accesses
Menus	Assigns	Assigns
Reports	Methods	Methods
SQL servers		
Windows		

Aside from the visual editor design entities, the remaining entities are created either by the developer or the system. You do not need to be concerned with the system-generated entities, since they are generated automatically.

Depending on the amount of customization you want to make to your interface design and the complexity of the event handler methods, about one-third to one-half of your development time may be spent on creating the visual editor-generated building blocks, while the remaining time can be spent developing your own custom code.

The next two sections describe developer-coded and system-generated entities in greater detail.

Developer-Coded Entities

Your part in the development process lies in the creation of methods—such as event handlers—that describe what should happen when an end user clicks on a push button or selects a menu command.

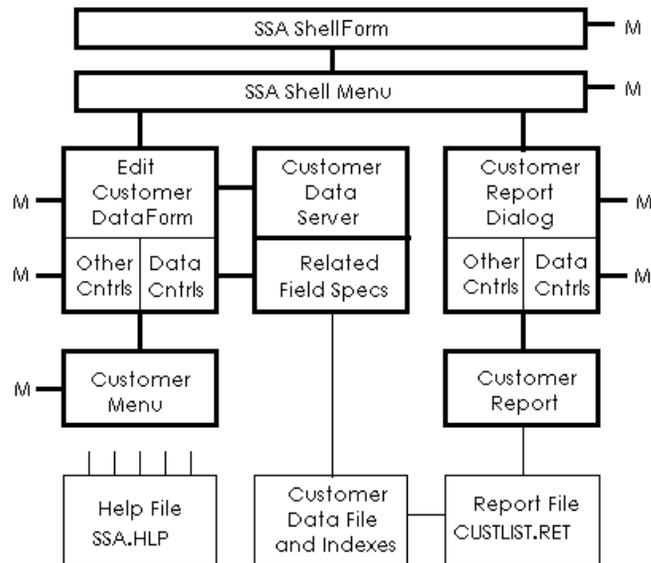
In the South Seas Adventures application, subclasses for some dialog windows are used to demonstrate how code reuse can greatly reduce your overall development time. (Subclassing is discussed in greater detail in [Chapter 7: Inheritance and Subclassing](#).) A few special window event handlers are also used, as you will see in [Chapter 10: Customizing Window Event Handlers](#).

System-Generated Entities

Entities that are generated automatically by CA-Visual Objects typically represent approximately 80 to 90 percent of the entities for an application like South Seas Adventures. The define entities make up about a third of the entities in the entire application. The vast majority of the other system-generated entities—including classes, access methods, and assign methods—are automatically generated by the IDE visual editors.

Linking the Remaining Building Blocks

The following diagram illustrates the entire structure of the South Seas Adventures application, including the supplementary building blocks, and the relationships among all the entities:



Key: M = method (event handler, button, or menu event)

Note: As noted earlier, rectangles with thick borders are the primary building blocks, while those with thin borders are external files. Design linkages are shown as thick lines, while external linkages are shown with thin lines.

This diagram expands upon the primary building block diagram, showing several additional relationships among the application entities.

The diagram shows the controls that you can place on each window when you are using the Window Editor. There are two types of controls—data controls, which could be linked directly to the fields in a data server (and, therefore, to a data file), and other controls, which are not data-aware. These other controls are used to accept user actions associated with push buttons and radio buttons or as inanimate display objects, such as group boxes, fixed icons, and fixed text.

Secondly, the important role played by field specs is shown as the rectangle adjacent to the data server. Note the design linkage to the data controls (via the name of the data server associated with a window). A dialog window may contain both types of controls, but there is no connection between a dialog's data controls to the fields in a data server and data file.

Lastly, you will notice the areas where the developer can explicitly code methods to describe how the application should act when the user makes certain control choices. These methods fall into the developer-generated entity category. Each place where a developer-coded method could be created is indicated by the letter "M." There are three types of such event-oriented methods:

- Push button methods

Can be added to indicate what should happen when the user clicks on a specific push button (refer to [Chapter 5: Creating and Using Windows](#) and [Chapter 6: Adding Controls to Your Windows](#) for more information).

- Menu event methods

Can be added to indicate what should happen when the user selects a specific menu command. As discussed in [Chapter 8: Creating Menus and Toolbars](#) each menu item has a related event, which can either be the name of a window, report, or method.

- Event handler methods

Can be added to any type of window. These methods are activated when the user takes certain actions on a window, such as clicking a mouse button or changing the contents of any edit control on the window (refer to "Customizing Window Event Handlers" for more information).

Begin Your South Seas Adventures...

Now, it is your turn to explore the world of CA-Visual Objects. The tutorial exercises take some time to work through, so try to complete a few chapters in each work session. Most importantly, you should complete the chapters in order, since certain exercises depend upon the building blocks created in earlier chapters.

Enjoy your adventure!

Exploring the CA-Visual Objects 2.7 Integrated Development Environment

This chapter demonstrates how to use the CA-Visual Objects 2.7 IDE to first import the South Seas Adventures application and then view its components. You will also learn how to build and run the application.

In addition, several CA-Visual Objects application design techniques will be discussed as follows:

- How to best organize files on your hard drive
- How to name your modules
- How to group entities into modules

Overview

Before you begin working on the tutorial exercises in this guide, there are some fundamental concepts that you should know about the way the components of a CA-Visual Objects application are organized and designed.

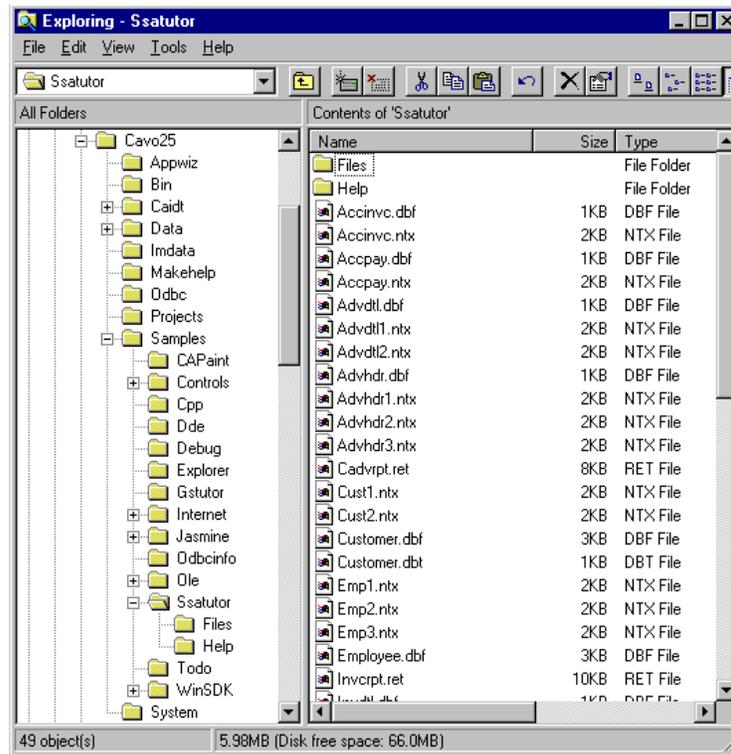
As a programmer, you know the importance of working with a development environment that is easy to manage. You must design a directory structure, define modules that hold all application entities, and be able to access them quickly.

Many of the details that commonly make your life difficult, in other programming environments, are automatically handled in CA-Visual Objects.

Choosing a Directory Structure

This section describes how the files related to the South Seas Adventures application are organized on your hard disk. Creating a separate subdirectory allows you to easily separate the various application components when you create install disks and backup sets. For the purposes of this tutorial application, a distinct subdirectory has been created. A more detailed description of the directory structure is presented later in this guide.

For example, the South Seas Adventures application was placed in a subdirectory called \SAMPLES\SSATUTOR. If you have installed CA-Visual Objects 2.7 to C:\CAVO27, the Windows Explorer will display the following directory structure:



The South Seas Directory Structure

This section describes each subdirectory within the South Seas Adventures application development environment.

SSATUTOR

This is the main runtime subdirectory. Any files necessary to run the application will be placed here, such as:

- The application executable, DLL, and help files (.EXE, .DLL, and .HLP, respectively)
- CA-QRT report files (.RET)
- Data files (for example, .DBF and .NTX)

SSATUTOR\FILES

This subdirectory is for resource files, such as bitmap (.BMP), icon (.ICO) and cursor (.CUR) files. Since these files are included in the executable file (.EXE or .DLL), your users do not really need the files in this subdirectory. You can also use this area to keep any application (.AEF) and module (.MEF) export files.

SSATUTOR\HELP

Creating context-sensitive help for the application is a project in itself. It can be easily segregated from the main development area. See the “Adding Help to Your Application” chapter for more information on creating help. The actual help file (SSA.HLP) associated with the application will be placed in the \SSATUTOR subdirectory.

The reasons for using this directory structure for the South Seas Adventures application are:

1. The application development directory structure resembles a user’s directory structure when the system is installed—resulting in a similar environment.
2. It is easy to create an installation set since the runtime files are kept in one place. The only other files needed by a user to run the application are the DLLs required by CA-Visual Objects 2.7. (For more information on DLLs, see [Chapter 18: Distributing Your Application](#) in this guide.

Creating Path-Independent Applications

CA-Visual Objects runtime file handling depends on the location of certain files (including database, report, and help files), as well as how you specify their locations when you create the server, report, and help building blocks.

To the extent possible, you should refrain from either specifying or hard coding any path information for these building blocks in order to maintain drive and directory independence for your application. There are also several steps you need to perform when you design entities using the CA-Visual Objects visual editors.

It is recommended that you refer to [Appendix A: Creating a Path-Independent Application](#) for more information related to path-independent applications.

The CA-Visual Objects 2.7 Multi-Tiered Repository

Many of the traditional development details are handled by the CA-Visual Objects 2.7 repository. The repository stores and organizes all of the application building blocks (or entities). It also manages other details of your application, including:

- Source code files
- Compiled code files
- Binary design components (such as windows and menus)
- Compiler options
- Directory information
- Dependency information

When you save an application, all the relevant information is stored in the repository files. This information can then be viewed or utilized while you are working in the CA-Visual Objects IDE.

The Repository Explorer provides you with a combination tree view and list view to browse projects, applications, libraries, DLLs, modules, entities, classes, and errors. It allows you easy access to the many parts that define the application.

Application Component Hierarchy

Application Level	The <i>application</i> is the top level in the hierarchy. GLOBAL variables are visible across the entire application.
Module Level	<i>Modules</i> exist on the second level of the hierarchy. Modules are similar in nature to traditional source code files, since they contain and encapsulate the supporting entities. STATIC GLOBAL variables are visible only within the module.
Entity Level	<i>Entities</i> are the smallest identifiable components of your application and are analogous to the functions, procedures, constants, and class definitions of traditional source files. The CA-Visual Objects repository manages each entity as a separate unit. LOCAL variables are visible to the creating entities only.

Automated Make and Entity-Level Compiling

The repository also keeps cross-reference information between the entities of your application. When you build your application, CA-Visual Objects knows which entities it has to compile, based on the changes you have made—only the entities that have changed will get recompiled. This is known as “entity-level” incremental compiling. Hence, you no longer need a traditional “make” file.

Grouping Your Entities into Modules

Module Design Considerations

As a programmer, you may often find yourself wondering how many modules are appropriate for the given application. There are a few general rules that may be helpful in this process, as described below:

1. The IDE visual editors place all source code entities related to a primary design entity in the same module as the design entity.
2. To avoid confusion, do not mix data servers, menus, and windows in the same module.

3. To avoid having too many entities in a module, restrict a module so that it contains no more than two or three windows or one menu, if possible. Although a question of personal choice, most modules should have between 20 and 100 entities.
4. Where possible, place all custom source-code entities (developer-coded) in a module other than the one in which the window or menu resides. You may have to cut and paste or drag-and-drop entities between modules to accomplish this.
5. Use descriptive names for the modules.

In the South Seas Adventures application, many of the module names begin the name as one of the business data types: adventure, customer, employee, invoice, item, and payment. The second part of the module name is a word such as class, data, windows, menu, methods, or reports. This naming convention indicates what is contained in the module. Note that the developer-coded methods are placed in the Methods modules.

Module Naming Conventions

This section discusses some of the tutorial modules in depth in order to demonstrate how the design considerations were applied. What you are striving for in your naming convention is ease of management in the future. Take advantage of long names for modules as well as entities. This will make the components of your application easier to recognize.

The names of the South Seas Adventures application modules are all two-part names separated by a colon (for example, Adventure:Data). The first part defines the primary focus of the entities in the module and the second part identifies for what the entities are used. For example, “:Forms” indicates that the module contains data windows, dialog windows, or a shell window.

To give you a practical idea of how these naming conventions are applied, let's now examine some of the Adventure modules:

Adventure:Data

This module contains all entities related to the two data servers associated directly with an adventure, the adventure header file (ADVHDR.DBF) and the adventure detail file (ADVDTL.DBF).

Adventure:Forms	<p>This module contains all the entities required for the adventure forms. The Window Editor was used to create these forms; therefore, the entities of the most concern within this module are naturally the window entities. There are actually five forms (window entities) in this module that let you add, edit, or browse data in the data servers mentioned above. These forms are named as follows:</p> <ul style="list-style-type: none">■ AdventureBrowser■ AdventureDetailSubform■ AdventureSubform■ EditAdventureWindow■ NewAdventureWindow <p>The AdventureSubform is a tabular display used on the AdventureBrowser window, and the AdventureDetailSubform is a tabular display of the items in the master-detail EditAdventureWindow.</p>
Adventure:Methods	<p>This module contains all the methods that were not generated automatically by the CA-Visual Objects editors. This code was separated from the generated code to make it easier to locate.</p> <p>In addition to modules that relate to one of the six business data types, you will also need some application-related modules. These contain entities used in several places. In the South Seas Adventures application, the general purpose modules include:</p>
Password:Forms	<p>This module contains the password dialog window and related entities.</p>
SSA Child:Menu	<p>This module contains the menu attached to the child windows.</p>
SSA Shell:Forms	<p>This module contains the shell window (SSAWindow) for the application.</p>
SSA Shell:Menu	<p>This module contains the menu attached to the shell window.</p>
App:Misc	<p>This module contains miscellaneous entities that can become a library or a DLL.</p>
App:Resources	<p>This module contains any cursors, bitmaps, and icons that the application needs.</p>
App:Start	<p>This module contains the application Start() method and any global entities for the application.</p>

Exercise

Now that you know the basics about how application components are handled and designed, you are ready for a hands-on tour of the IDE. You will discover the ways it can help you work with your application—during various stages of development.

In the following exercise, you will open the CA-Visual Objects 2.7 IDE, import the South Seas Adventures application, set some application properties, compile the application, and then run it.

Using the Repository Explorer

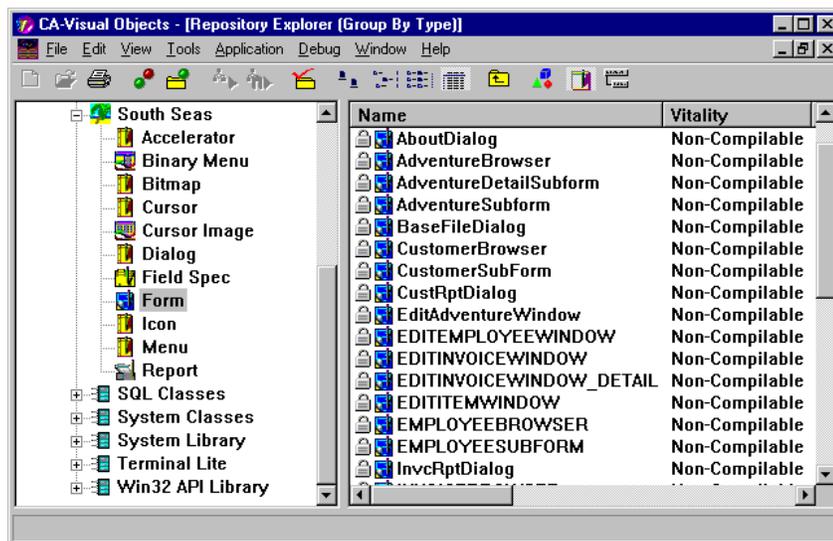
The Repository Explorer provides you with a logical way to access and view the components of your application. There are several ways to view your applications in the CA-Visual Objects 2.7 Repository Explorer, including the tree view and the list view.

Let's look at the Repository Explorer by performing the following steps:



1. Start CA-Visual Objects 2.7 from its Windows folder by clicking on the Start Menu and selecting the Programs folder. After selecting the Programs folder, choose the CA-Visual Objects 2.7 folder and select CA-Visual Objects 2.7.

The Repository Explorer is displayed showing the tree view and list view:



Initially, the Repository Explorer displays the Default Project and all of the libraries supplied with CA-Visual Objects 2.7. Each library is represented by a branch in the tree view. Similarly, the applications and libraries that you create also appear as branches on the tree.

Note: The Lock icon in the Name column is only displayed if Visual Source Safe is installed on your machine.

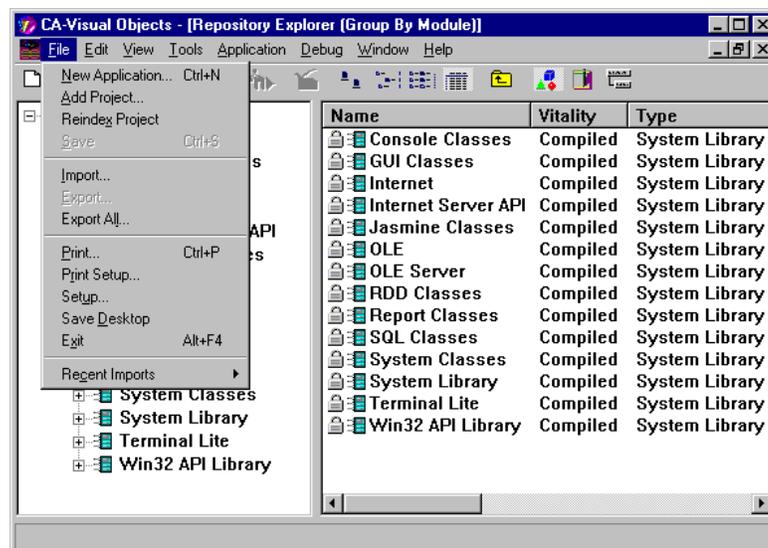
- Notice the CA-Visual Objects 2.7 default icons that are displayed on the Repository Explorer. These distinctive icons help you visually distinguish between applications and libraries.

The following table shows the default CA-Visual Objects icons:

Icon	Description
	CA-Visual Objects System library icon
	Application icon (which you can create)
	User-defined library icon
	User-defined DLL icon

You can also create your own application icon, which you will learn more about in [Chapter 11: Working with Icons and Cursors](#) in this guide.

- The Repository Explorer allows you to create, update, and manage your applications easily. Selecting the File menu will allow you to create new applications and projects or import and export existing applications:



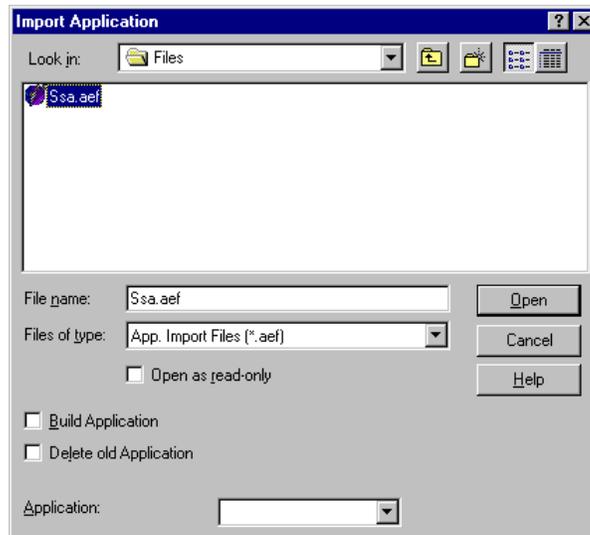
The Import and Export commands are provided to allow you to move applications to and from external disk files, since applications are stored in the repository. You will find these options useful for trading applications with other developers and for backup purposes.

Importing the Application

Let's begin by importing the South Seas Adventures application, which is used throughout the remainder of this tutorial:

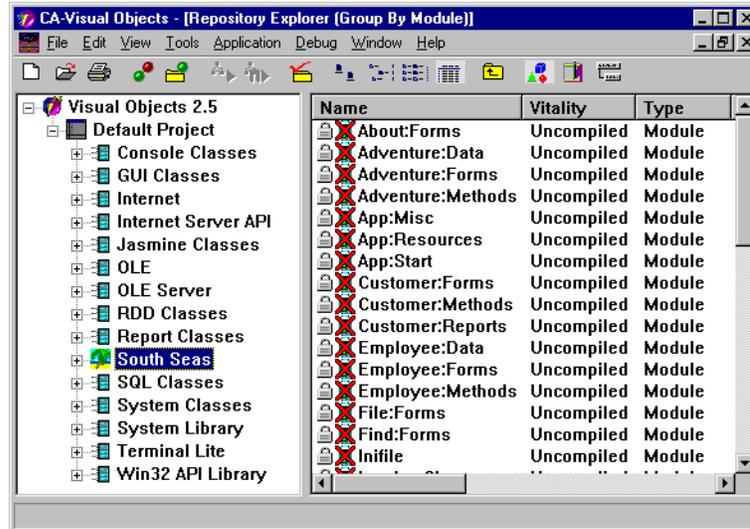
1. Select the Default Project branch from the Repository Explorer tree view.
2. Choose the Import command from the File menu.

The Import Application dialog box appears:



3. Select SSA.AEF. This is the South Seas Adventures application export file. It can be found in your CA-Visual Objects 2.7\SAMPLES\SSATUTOR\FILES subdirectory.
4. Choose OK.

After the application is imported, a new branch appears on the Repository Explorer tree view. This branch represents the South Seas Adventures application, as shown below:



Notice the palm tree icon on the South Seas Adventures branch:



This icon is associated with the application through the Application Options dialog box, which you will learn about in the next section.

Configuring Your Application Environment

For the purposes of the tutorial, there are a few settings that you may have to modify, based on how you installed CA-Visual Objects 2.7. You will also look at the compiler options that are available to you.

Application Options

As suggested previously, you should create a subdirectory for each application or library that you create. This ensures that your disk stays organized, by keeping files for each application separate, and eliminates confusion about what files belong to a specific application. The path for EXE and DLL files can then be set to the application's subdirectory.

You can modify the properties of an application or library at any time during development. The Application Properties dialog box allows you to specify the following:

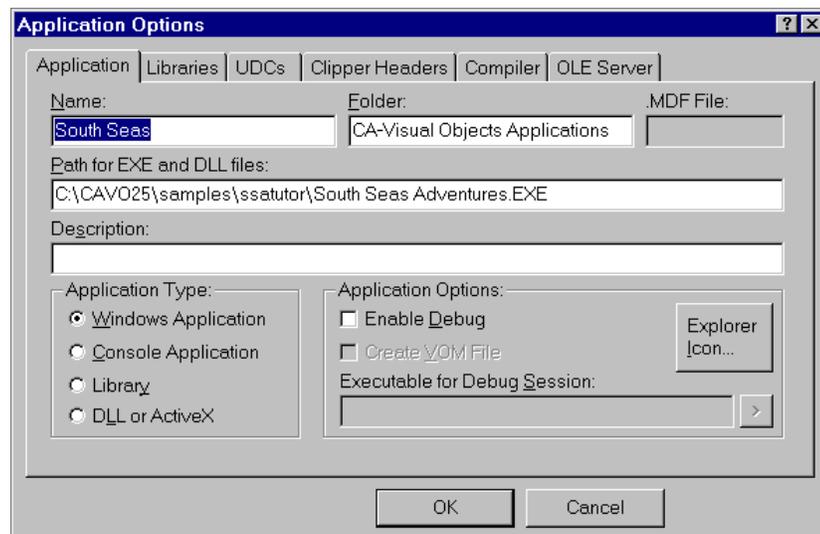
- Application name
- Application type
- Libraries to include in your search path
- Path for a generated file (.EXE or .DLL)
- Folder in which to place the generated .EXE file
- Default debugging status
- Whether the generated .EXE or .DLL file should include CA-Visual Objects 2.7 runtime files

Now, let's see how this applies to the South Seas Adventures application:



1. Select the Application Properties toolbar button or right-click on the application and select Properties from the local pop-up menu.

The Properties of South Seas dialog box appears:



2. In the Path for EXE and DLL Files edit control, redirect the executable to the SSATutor directory by setting the path to:

```
%CAVOSAMPLESROOTDIR%\SSATUTOR\SSA.EXE
```

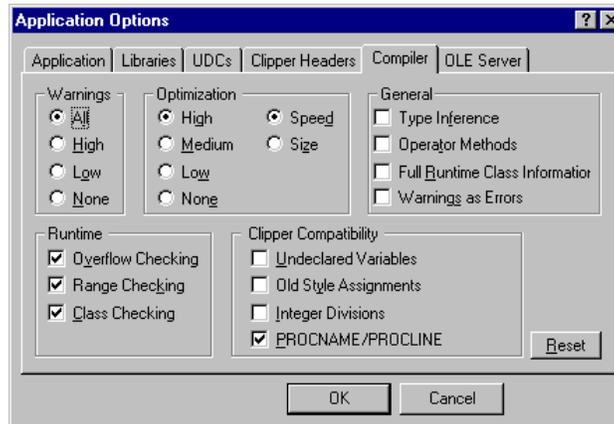
Compiler Options

Before attempting to compile the application, it is wise to review its compiler options. You can specify different compiler options for each application at any time during the development process.

Let's now look at the compiler options for the South Seas Adventures application:



1. Select the Compiler Options tab in the Application Options dialog window to view the compiler options available to you:



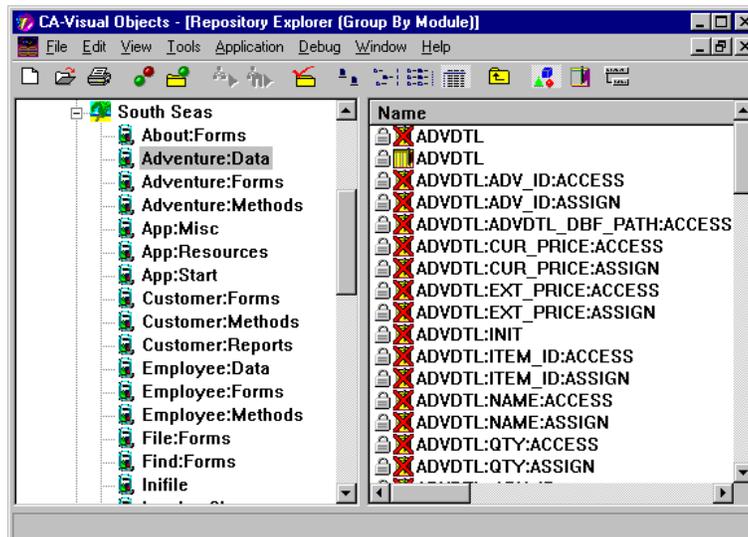
2. Set the Compiler Options. This is accomplished by selecting each item's check box or radio button.
3. Choose OK to close the Application Options dialog box.

Viewing the Application Modules

In the following exercise, you will expand the CA-Visual Objects Repository Explorer to examine the entities in a module, and use the list view to look at the various types of entities.

1. Open the South Seas Adventures application by double-clicking on its branch on the Repository Explorer.
2. Double-click on the Adventure:Data module to examine its entities.

The list view displays the entities in the module:



Note: The red X denotes that the entity needs to be compiled. We will compile all of the entities in the following [Building the Application](#) section.

There are two server entities (AdvHdr and AdvDtl), each associated with many field spec entities. Further down in the list, you will find the two data server classes and related methods, accesses, and assigns.

3. Scroll through the list, and double-click one of the two Init() methods and a few access and assign entities to see the kind of source code that is generated by the DB Server Editor.

Building the Application

Once you have imported (or finished developing) your application, it can be easily compiled. To build the South Seas Adventures application:

1. Click on the South Seas Adventures application branch on the Repository Explorer tree view.
2. Click the Build button on the toolbar.



Note: When you import an application, none of the entities are compiled. The South Seas Adventures application could be considered a medium-sized application, so this initial build could take some time. Subsequent builds will be much faster, since only those entities that need compiling, based on your changes, will be compiled.

Once your program compiles successfully—indicated by the word Compiled in the Vitality column of the List View and by the removal of the red X from each entity—you can run your program. The next section discusses how to do this.

Running the Application

During development, you can either run your application dynamically from the IDE or you can generate an .EXE file and run it from Windows.

Regardless of the option you choose, when the application is run, the current directory will be set to the path specified in the Application Options dialog box. This allows you to test both versions from the same directory.

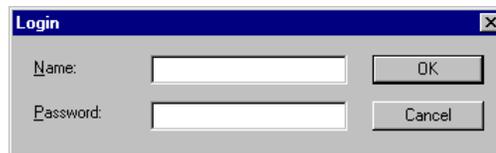
Running the Program Dynamically

As you incrementally develop your application, and then compile your latest changes, you can run it without even having to leave your IDE environment. To do this, perform the following steps:

1. Click on the South Seas Adventures application branch on the Repository Explorer Tree View:
2. Click on the Execute toolbar button to run the program from the IDE.
3. Choose OK in the South Seas Adventures opening dialog box:



The Login dialog box appears:



4. You can enter the system by logging in using the following procedure:
 - Type **User** in the Name edit control.
 - Type **Trainee** in the Password edit control.
 - Choose OK.
5. Close the South Seas Adventures application by double-clicking the system menu and select Yes when prompted.

Creating and Running an Executable File

Typically, after you have finished development, you will want to create a stand-alone executable program (.EXE) which can be run without the CA-Visual Objects IDE.

To create the .EXE file and run this program, perform the following steps:

1. Click on the South Seas Adventures application branch on the Repository Explorer Tree View.



2. Click on the Make EXE toolbar button.

Once the .EXE file has been generated, your application will be in the folder specified in the Application Options dialog box.



3. You may now run the program by clicking the South Seas Adventures application item in the folder.

The opening dialog box appears.

4. Choose OK.

The Login dialog box appears.

5. You can enter the system by logging in using the following procedure:

- Type **User** in the Name edit control.
- Type **Trainee** in the Password edit control.

6. Choose OK.

7. Close the South Seas Adventures application by double-clicking the system menu and select Yes when prompted.

Summary

This chapter provided a brief overview of the IDE, as well as some useful tips to make your application development easier. If you need more information on using and navigating through the IDE, see the *CA-Visual Objects 2.7 IDE User Guide*.

Working with Data Servers

This lesson introduces you to the basic concepts of data servers. You will:

- Create a Customer data server based on the Xbase model of a database file (.DBF) and two SQL data servers
- Use the data servers in data windows
- Look at server notification
- Explore programming techniques used to implement data servers

Overview

Data servers are the object-oriented means by which your applications communicate with databases. CA-Visual Objects 2.7 applications can communicate with DBF databases and SQL databases. These database formats are very different and, in other programming languages, require drastically different approaches when writing applications.

CA-Visual Objects overcomes this difficulty through data servers. The `DataServer` class is the base class from which all data servers are derived. It defines a common set of methods and properties, based on a common database paradigm, which all server objects use. This means that you no longer have to code specifically for a particular data model.

The following table displays the data server methods that are designed to be compatible with the Xbase DML (data manipulation language):

Xbase (CA-Clipper)	Data Server Classes
USE Customer ALIAS Cust NEW	NEW oCust := Customer{}
DO WHILE CUST->!EOF()	DO WHILE !oCust:EOF
IF Cust->Sex == "M"	IF oCust:Sex == "M"
Cust->(DBDelete())	oCust:Delete()
ELSE	ELSE
Cust->Salary += ;	oCust:Salary +=;
Raise(Cust_Name)	Raise(oCust:Name)
ENDIF	ENDIF
Cust->(DBSkip())	oCust:Skip()
ENDDO	ENDDO
USE	oCustServer:Close()

Two editors, the DB Server Editor and the SQL Editor, are used to create data servers. The DB Server Editor creates a class which is derived from the DBServer class. The DB Server Editor will also create the associated entities to accompany this class creation. This class is used to access DBF-type databases. The SQL Editor creates a class (and associated entities) which is derived from the SQLTable class. This class is used to access SQL databases via ODBC.

The main difference between these classes is in the way you instantiate them. Also, each data server contains methods and properties specific to the type of database they serve.

Exercise

In this lesson, you will create the data servers used by the South Seas Adventures application to access customer and invoice data.

Creating a Customer Data Server

At one time, South Seas Inc. relied on a customer-tracking system written in CA-Clipper for the character mode environment. Now, they wish to incorporate the data from that system into the new South Seas Adventures application in Windows.

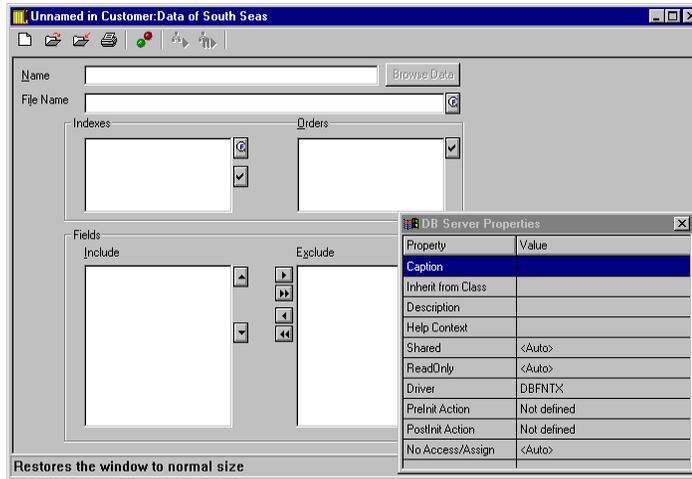
Invoking the DB Server Editor

You are now going to create the data server from an existing .DBF file, using the DB Server Editor:

1. Open the South Seas Adventures application by double-clicking on its branch on the Repository Explorer.
2. To create the data server in a new module, choose the New Module toolbar button—the Create Module dialog box appears.
3. In the Enter module name edit control, type **Customer:Data** and choose OK. This adds the Customer:Data branch to the South Seas Tree View on the Repository Explorer.
4. Select the Customer:Data Module.
5. Select the DB Server Editor command from the Tools menu (or click on the Open Entity toolbar button and choose DB Server Editor from the local pop-up menu).



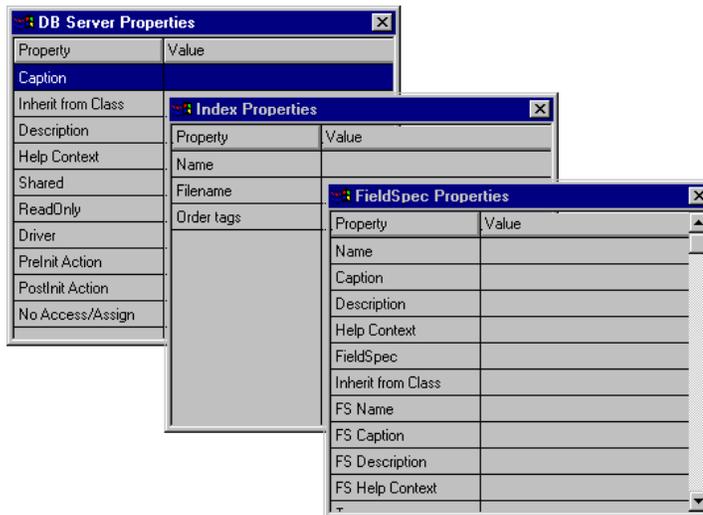
The DB Server Editor is displayed. It consists of a workspace and a floating Properties window:



Initially, the Properties window displays properties associated with the server as a whole, as indicated by the title "DB Server Properties."

The Properties window changes depending on which edit control is active in the editor workspace. For example, when the active control is the Indexes list box, the Properties window displays Index properties. If the active control is a field in the Include list box, the Properties window displays field specification properties.

Each of these windows is displayed below:



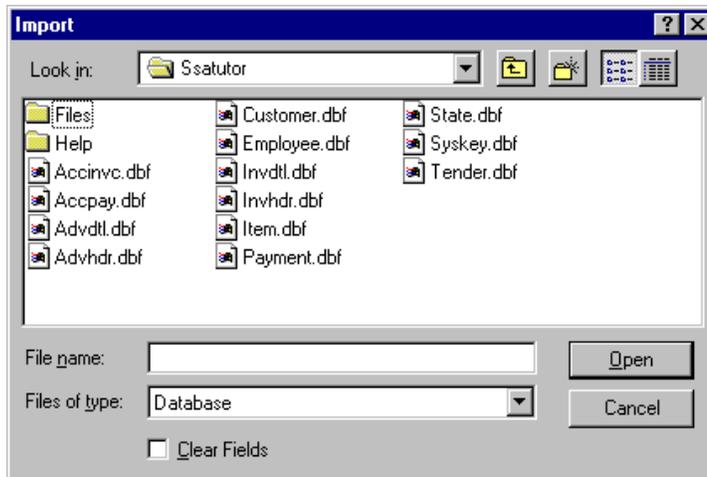
Importing a .DBF File

The DB Server Editor allows you to create a data server from scratch, or from an existing .DBF file. The CUSTOMER.DBF file already exists, so let's use the Import feature to read in its structure:



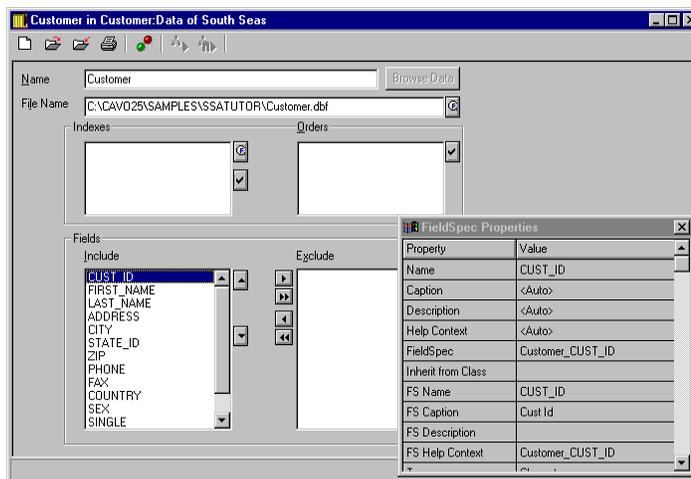
1. Choose the Import command from the File menu (or click on the Find button located to the right of the File Name edit control).

The Import dialog box is displayed:



2. Select the CUSTOMER.DBF file located in the CA-Visual Objects 2.7 SAMPLES\SSATUTOR subdirectory and choose OK.

The Import feature fills in the Name and File Name edit controls, as well as the Include list box control in the Fields group box:



The file name is placed in the Name edit control. This name serves as the data server name and the class name for the data server. It acts as a prefix to the classes created for each field in the database.

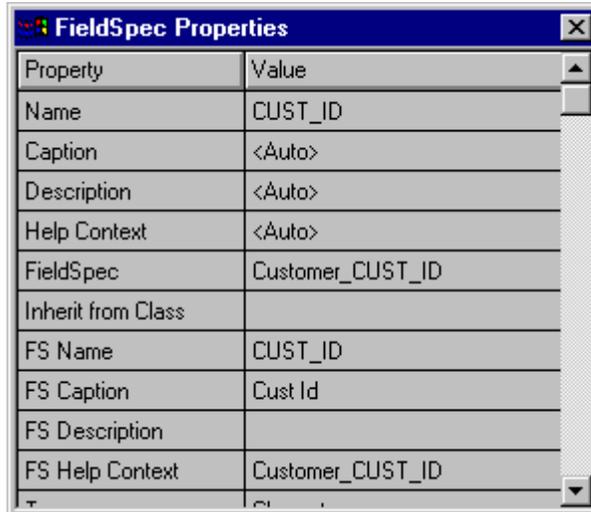
The physical character mode file name, including the full path, is placed in the File Name edit control.

Important! If you choose to leave the full path as is, the resulting application attempts to find the Customer database in that directory. This limits the installation options for your application. Refer to [Appendix A: Creating a Path-Independent Application](#), for information on how to modify this path at runtime.

3. Remove the drive and directory information from the File Name edit control, leaving only CUSTOMER.DBF.
4. The Include list box is populated with the names of the fields in the Customer database. The fields are listed in the order in which they are defined in the database. Select the CUST_ID field from the Include list box:



The Properties window now displays the properties for the Cust_ID field:



Each field of a data server has a field specification which is made up of many properties.

Tip: Scroll down through the FieldSpec Properties window to see the various properties.

The Include and Exclude list boxes allow you to control which fields are accessible through this data server. Fields in the Exclude list box are inaccessible while using this server.

Importing an Index

Now, let's select the index files for use in the South Seas Adventures application:



1. Click the Find button located to the right of the Indexes list box.

The Browse dialog box displays.

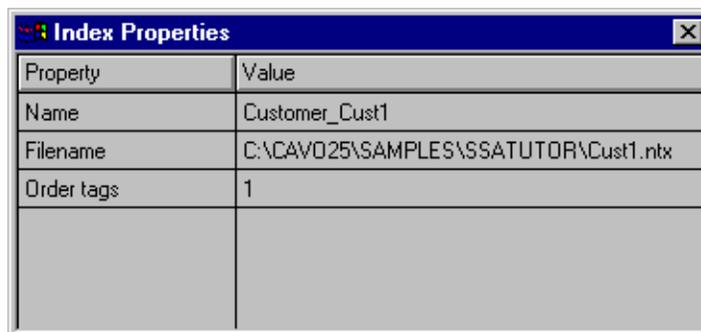
2. Select the CUST1.NTX file located in the CA-Visual Objects 2.7 SAMPLES\SSATUTOR directory and choose OK.

CUST1.NTX now appears in the Indexes list box:



The check mark indicates that this is the *controlling index*, which means that it contains the order used to control the logical order in which the database file will be processed.

Notice that the properties window now displays the Index Properties:



These properties specify the index file name and the number of orders within the index.

- Click the Filename property and remove the drive and directory information, leaving only CUST1.NTX.

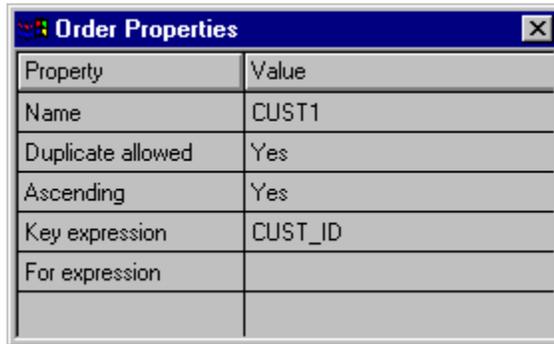
In order to support multiple index files (such as .CDX and .MDX), the index orders are displayed in the Orders list box, with the *controlling order* also indicated by a check mark:



Since .NTX files can only support one order, no more than one order can be added to this list.

- Click the CUST1 order.

The properties window now displays the Order Properties of the CUST1.NTX index:

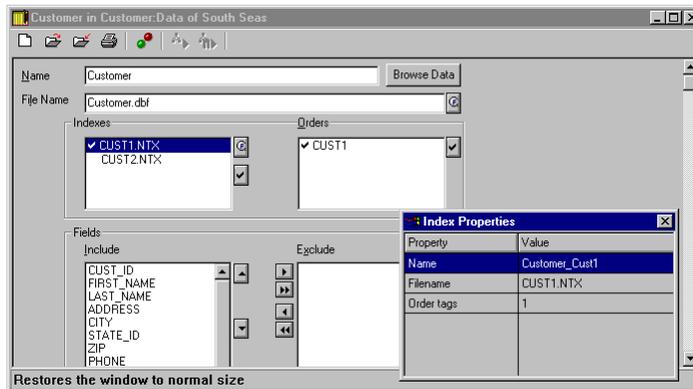


This is where you define the properties of the index order. The CUST1.NTX index will be in ascending order by the contents of the Cust_ID field.

- Repeat steps 1 through 3 for the CUST2.NTX index file.

Remember to remove the paths from each of the index file names.

When all these steps are completed, the DB Server window should look as follows:



Saving the Data Server



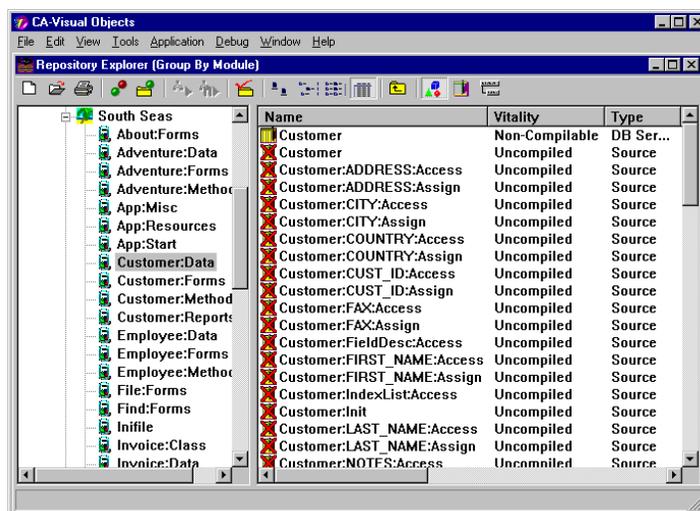
Now, you are ready to save the data server:

1. Save the data server by clicking the Save toolbar button.

CA-Visual Objects 2.7 now creates the entities required for your program. Watch the DB Server Editor's status bar as it creates these entities.

2. Close the DB Server Editor by double-clicking its system menu.
3. To see the entities created, open the Customer:Data module by clicking its branch.

The Repository Explorer displays the entities in the Customer:Data module:



4. Scroll through the entities in the Customer:Data module to see what was created when you saved your work.

There are many entities related to the data server. Each field has a `FldSpc` binary entity, a `FieldSpec` subclass, an `Init()` method, an access method, and an assign method. For the data server, there is a server entity, a `DBServer` class, an its `Init()` method.

Creating a SQL Server

The South Seas Adventures application serves as a front-end management and point-of-sale tool. The invoices and payments created using this system must be sent to the Accounting department.

For purposes of this tutorial, assume that the Accounting department of South Seas Inc. uses a separate system on a SQL database. This database is accessible via Open Database Connectivity (ODBC).

CA-Visual Objects provides ODBC drivers for many major databases.

If you did not select all the components when you originally installed CA-Visual Objects 2.7, refer to the [Installing ODBC Drivers](#) section in this chapter.

To simulate access to the Accounting department's data, you will access .DBF files via the dBASE ODBC driver provided with CA-Visual Objects.

Tip: This is a great way to prototype applications which will eventually be connected to a SQL database. For development purposes, you can use the dBASE ODBC driver. When creating your .DBF files, make the DBF layout the same as the SQL accounting database. For the final release, simply modify the SQL server to use the new data source and recompile.

Installing ODBC Drivers

The CA-Visual Objects 2.7 installation program allows you to install any number of the available ODBC drivers. For this lesson, you need to have the dBASE ODBC driver installed. If you already installed this driver, skip to the next section, The ODBC Administrator.



To install the dBASE ODBC component on your hard drive:

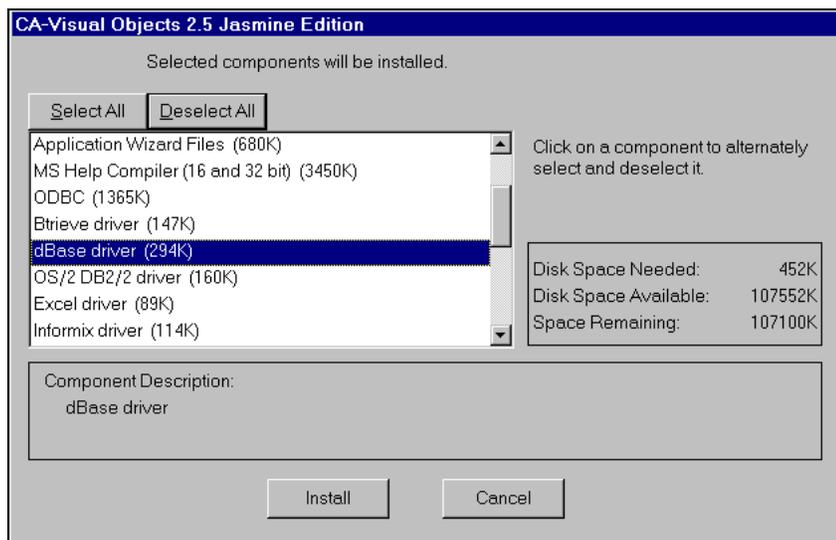
1. Insert the CA-Visual Objects CD-ROM in the CD-ROM drive.
2. For Microsoft Windows or NT, click the Start button and then click Run.
3. In the Command Line box, enter:

```
cd-rom_drive:\setup
```

Where *cd-rom_drive* represents the drive letter of the CD-ROM drive—for example, you might type **e:\setup**.

4. Choose OK.
This will invoke the CA-Installer. A text window displays containing important information regarding the installation of CA-Visual Objects.
5. Choose OK.
The Registration window displays. The Registration information is required in order to proceed with the installation.
6. Enter your Registration information and click OK.
7. A window appears containing the default directory where CA-Visual Objects will be installed. You can specify another directory by typing in the full path.

8. Choose Continue.
- The Selected Components Window appears.
9. Click the Deselect All push button.
10. Scroll through the available options, and click the dBASE driver component:



12. Click the Install push button.
13. The CA-Installer will proceed to install the dBASE ODBC driver. Simply follow the on-screen prompts to proceed with the installation.

The ODBC Administrator

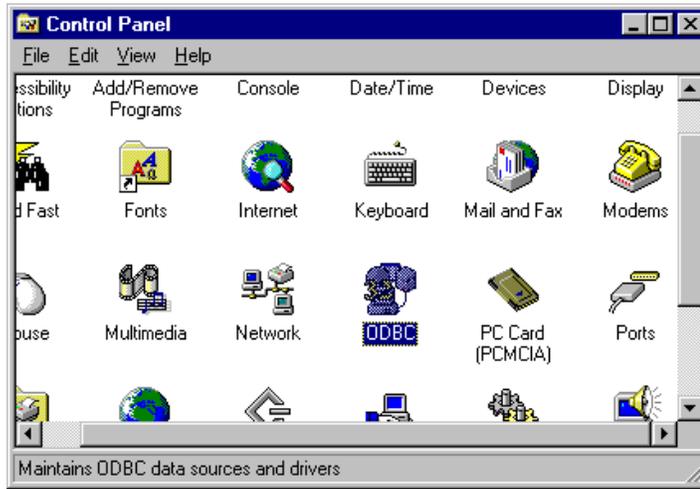
Before you use the SQL Editor to define a SQL table for the first time, you must define the data sources using the Windows ODBC Administrator.

The ODBC Administrator allows you to add, delete, or configure data sources. A data source is the data you want to access and the information needed to get to that data.

The ODBC Administrator is accessed through the Windows Control Panel, as follows:

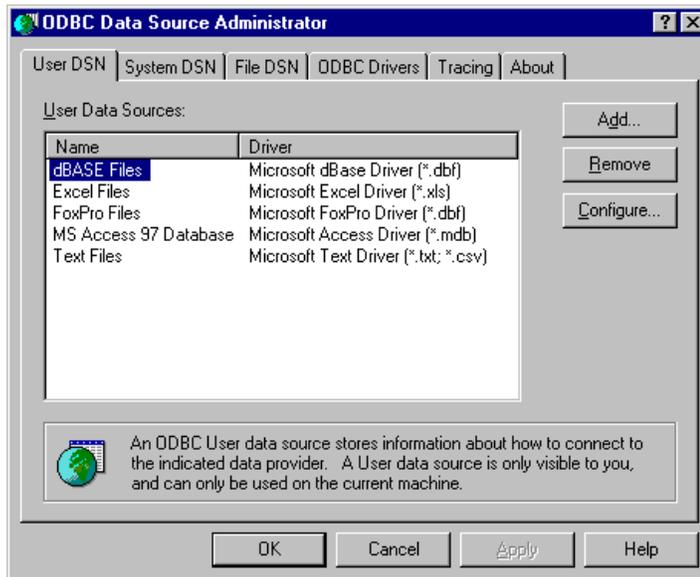
1. Click the Start Button.
2. Highlight the Settings Folder.

3. Click the Control Panel icon. The window is displayed:



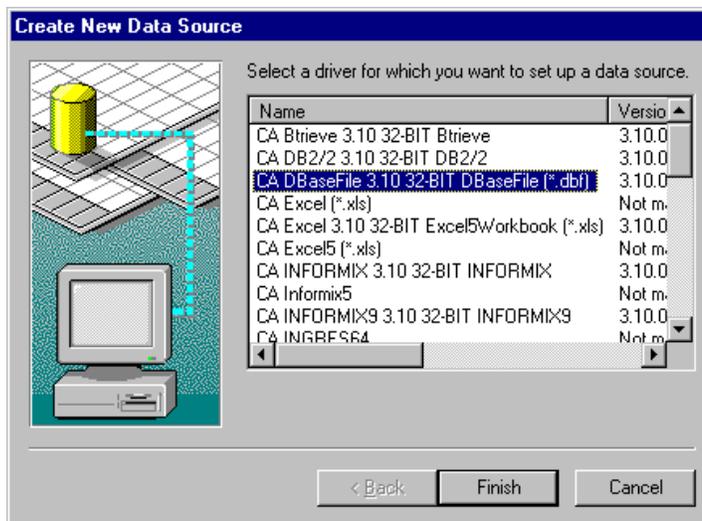
Note: For Windows users, the name of this icon is “32-bit ODBC.” For Windows NT users this icon is named “ODBC.”

4. Double-click the 32-bit ODBC program icon to launch the ODBC Administrator. The Data Sources dialog box appears:



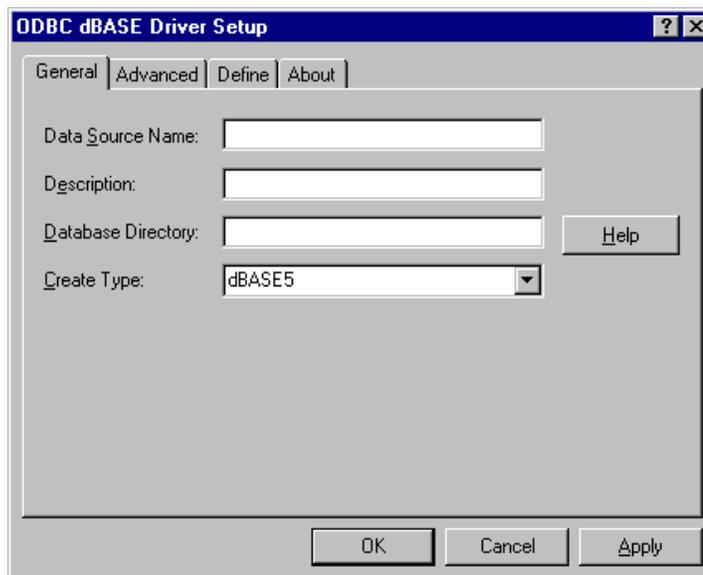
5. Add a new data source by clicking on the Add button.

The Add Data Source dialog appears, displaying all of the drivers that you have installed:



6. Select CA DbaseFile 3.10 32-BIT DBaseFile (*.dbf) from the Installed ODBC Drivers list box, and then click Finish.

The ODBC dBASE Driver Setup dialog box appears:



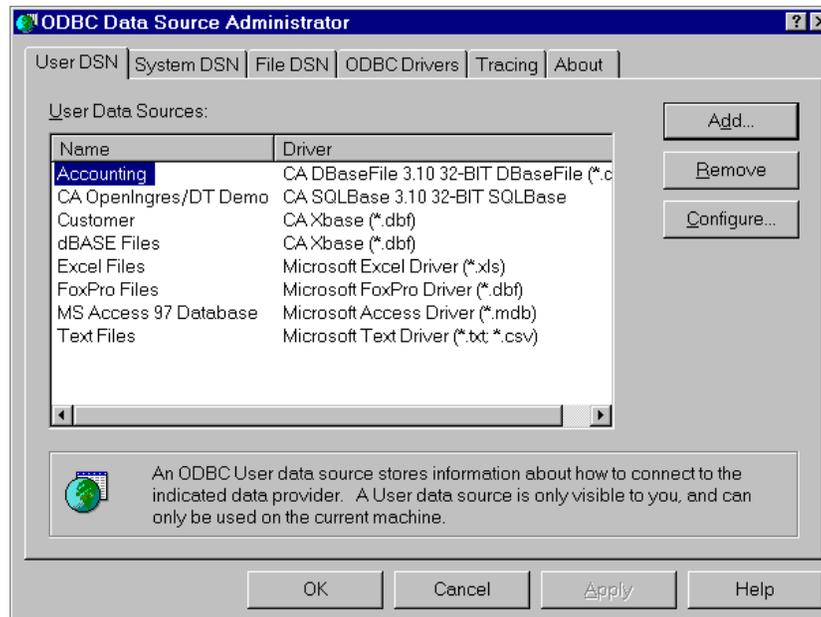
7. In the Data Source Name edit control, type **Accounting**.

This is the name that is searched for when choosing a data source from inside of the SQL Editor.

8. In the Description edit control, type **Accounting Department Data**.

9. In the Database Directory edit control, type in the path to the SAMPLES\SSATUTOR subdirectory, which is located in the CA-Visual Objects 2.7 installed directory (for example, C:\CAVO27\SAMPLES\SSATUTOR).
10. From the Create Type drop-down list box, select Clipper.
This ODBC driver allows access to all Xbase data.
11. Change to the Advanced tab and confirm that the Locking is set to RECORD and the Lock Compatibility is set to Clipper.
You are now finished defining the data source.
12. Choose OK.

You should see your new entry in the Data Sources (Drivers) list box when the Data Sources dialog box reappears:



13. Choose the OK push button, and then close the Control Panel by double-clicking on the system menu.

Using the SQL Editor

Now, you are ready to use the SQL Editor:

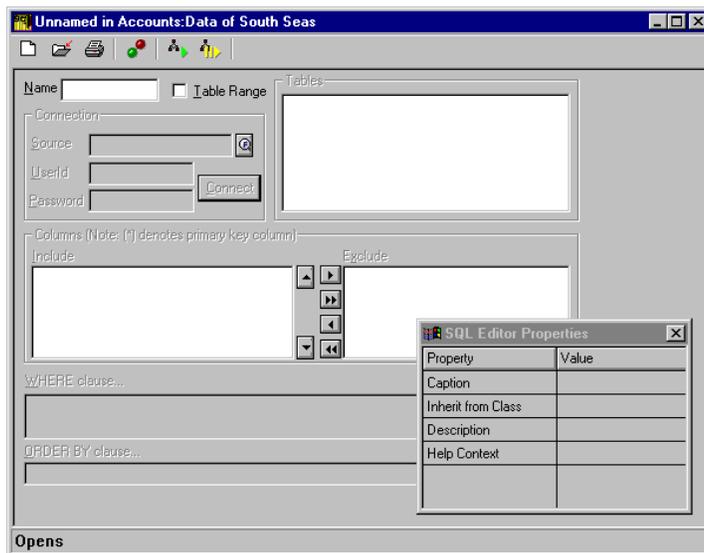
1. Open the South Seas Adventures application (if not already open) by double-clicking its branch on the Repository Explorer tree view.
2. To create your SQL server in a new module, select the New Module command from the File menu or click the New Module toolbar button.
3. Name the new module **Accounting:Data** and choose OK.





4. Select the Accounting:Data module.
5. Click on the Open Entity toolbar button and select SQL Editor, or select the SQL Editor command from the local pop-up menu.

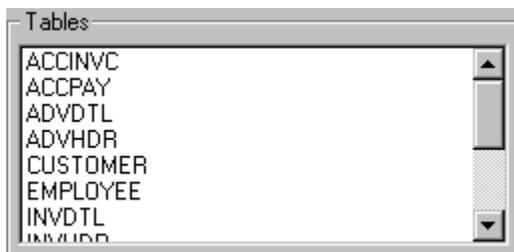
The SQL Editor window appears:



6. In the Name edit control, type **ACCINVC**.
 The SQL Editor uses this name to create the AccInvc class, which you are going to use later to access the data from your program.
 Now that you have a name, you may select the data source that the AccInvc server accesses.



7. Click the Find button, which is located to the right of the Source edit control.
 The SQL Data Sources dialog box displays.
8. Select the Accounting data source from the Machine Data Source tab and click OK.
 The Tables list box is now populated with a list of the .DBF databases or tables that are in the directory you specified for the Accounting data source.
9. Select the ACCINVC table from the list:



This is the destination file for the invoice information sent by the South Seas Adventures application.

The SQL Editor fills the Include list box with the columns (fields) of your table. As with the DB Server Editor, you can choose to exclude columns from the server. Excluding columns has no affect on the actual table, they simply are not accessible by the server that excludes them.

Field spec entities have also been created for each column. Since you are not going to use this server as part of a window, the default field specifications need no modification.

If you require the table rows to be filtered, you can add an SQL WHERE clause in the WHERE clause multi-line edit control. You can also specify an SQL Order clause to sort the rows.



10. For the purposes of this lesson, this SQL Server definition is complete. Therefore, select Save from the File menu.



11. Select the New Server toolbar button to clear the current editor.
12. Repeat steps 5 through 10 to create the AccPay server. The AccPay table also resides in the Accounting data source.
13. Once this is done, exit the SQL Editor by double-clicking on its system menu.

Programming with Servers

You will now look at a support method that demonstrates how a data server is typically used. It actually implements the transfer of data from the South Seas Adventures application to the Accounting Department.

Importing a Support Module

First, you must import a support module export file (.MEF), containing the predefined OptionsSubmit() method, using the following steps:

1. Select the South Seas application and choose the Import command from the File menu.
2. From the Import dialog box, select the files of type combo box and select Mod.Import files (*.mef). After choosing the .MEF files, select the TUTSERV.MEF file located in the SAMPLES\SSATUTOR\FILES subdirectory.
3. Choose Open.

Notice that a new module called Tutorial:Servers has been added to the South Seas Adventures application.

Viewing the Server Source Code

View the source code using the following steps:

1. Open the Tutorial:Servers module by clicking its branch on the Repository Explorer.
2. Open the Source Code Editor by double-clicking on the OptionsSubmit() method of the SSAWindow class in the Repository Explorer list view.

Notice that the code uses the AccInvc and AccPay SQL servers in the same way that data servers based on .DBF files are used. In fact, had you not just created the servers, you would probably not know what type of server they are.

In the OptionsSubmit() method, you will find two loops—one for processing invoices, the other for processing payments.

The payment processing loop proceeds as follows:

1. Define and create the necessary data server objects:

```
LOCAL oAccPay AS AccPay
LOCAL oPayment AS Payment
oAccPay := AccPay{}
oPayment := Payment{}
```

2. Position the Payments data server at the first record:

```
// Submit payments
oPayment:GoTop()
```

3. Loop while there are records to process and the user wants to continue. This method uses a progress bar dialog box that allows the user to cancel the process:

```
DO WHILE !oPayment:EOF .AND. ;
!oProgressDialog:CancelRequested
```

4. Check to see if the current payment record has already been submitted to the Accounting department:

```
IF !oPayment:Submitted
```

5. If the current record has not been submitted, add a record to the Accounting department's database and update its fields:

```
oAccPay:Append()
oAccPay:Inv_ID := oPayment:Inv_ID
oAccPay:Pay_Date := oPayment:Pay_Date
oAccPay:Tender_ID := oPayment:Tender_ID
oAccPay:Amount := oPayment:Amount
oAccPay:Details := oPayment:Details
oAccPay:Expiry := oPayment:Expiry
oAccPay:CardNo := oPayment:CardNo
```

6. Force a write to the database:

```
oAccPay:Commit()
```

7. Set the payment record as having been submitted:

```
oPayment:Submitted := TRUE
```

8. Advance the progress bar:

```
oProgressDialog:Advance;  
("Reviewing Payment #: "+ ;  
oPayment:Inv_ID)
```

9. Go to the next record in the payment record and return to top of loop:

```
oPayment:Skip()  
ENDDO
```

10. Close all files:

```
oAccPay:Close()  
oPayment:Close()
```

Running the Application

Now, let's see the code in action:



1. You must first build the application by clicking the Build toolbar button.



2. Run the application by clicking on the Execute toolbar button. Click Continue when the pop-up information screen appears.

3. At the Login dialog box, type **User** in the Name edit control and **Trainee** in the Password edit control, and choose OK.

4. Select the Submit Invoices and Payments command from the Options menu.

All unsubmitted invoices and payments currently in the system will be sent, via the ODBC connection, to the Accounting Department's database. At this point, there are no invoices or payments in the database, so no data will be sent. The Submission Report dialog box informs you how many records have been submitted.

5. Close the Submission Report dialog box by clicking OK.

6. Close the South Seas Adventures application by choosing Exit from the File menu and then selecting Yes when prompted.

Event Notification

Up to this point, you have seen data servers as simple tools for programming. But, they are capable of much more with minimal coding.

Client Data Forms

Data servers are aptly named. Within an application, each data server object has clients, in particular—data windows.

A data server will notify its client data windows of all operations affecting the data server. This allows the data windows to keep themselves up to date with respect to the data server, for example, updating the data display, appending records, and moving the record pointer.

To attach a data server to a window, all you need to do is perform a Use() operation on the data server.

1. Open the Employee:Forms module by clicking its branch on the Repository Explorer.
2. Find the EditEmployeeWindow:Init() method in the Repository Explorer List view and open it by double-clicking on it. The following code appears in the Source Code Editor:

```
METHOD Init(oWindow, iCtlID, oServer);
    CLASS EditEmployeeWindow
    LOCAL oIserver AS OBJECT
    SELF:PreInit (oWindow/iCHID, oServer, uExtra)
    SUPER:Init(oWindow, ResourceID ;
        {"EditEmployeeWindow"}, iCtlID)
    ...
    IF (oServer = NIL)
        SELF:Use(Employee{})
    ELSE
        SELF:Use(oServer)
    ENDIF
    ...
    SELF:ViewAs (#FormView)
    SELF:PostInit (oWindow, iCHID, oServer, uExtra)
    RETURN SELF
```

This method was created when the EditEmployeeWindow window was saved from within the Window Editor. This method performs a Use() on either a data server passed to the window, or the Employee data server defined for the window in the Window Editor.

SELF:Use(Employee{}) registers the window as a client of the Employee data server. When something happens, such as record pointer movement to the EditEmployeeWindow object's attached Employee data server, the window is notified. The data window then takes the appropriate action—such as updating its controls.

What if a data server has two client data windows? Both will be notified when something happens to the data server. If data changes in one data window, the other is notified and changes are automatically displayed.

Child Servers

A data server can also be related to another data server via a SetRelation or SetSelectiveRelation link. The server issuing the SetRelation or SetSelectiveRelation call becomes the parent, while the other becomes the child.

Any movement in the parent server automatically causes movement in the child. In addition to movement actions, the `SetSelectiveRelation` link limits visible records in the child server to those that match the relation as demonstrated in the following paragraphs.

Note: `SetSelectiveRelation` links are used on data windows that contain subform controls. The subform control has its own attached data server. The data server of the window is the parent, while the data server of the subform control is the child.

An example of this can be found in the `Init()` method of the `EditAdventuresWindow` class in the `Adventures:Forms` module. This code was generated automatically using the Master Detail option of the Auto Layout feature in the Window Editor. The `EditAdventureWindow` deals with two data servers. It uses them in a master-detail relationship.

In this case, the master server is related to the detail server using a `SetSelectiveRelation()`. Using this type of relation filters the child server so that only those records that match the relation key are visible. In the `EditAdventureWindow` window, you only want to see the detail records of a particular adventure.

The `EditAdventureWindow` contains a subform called `AdventureDetailSubform` (see the `Adventure:Forms` module). The subform server, `AdvDtl`, is specified at at line 53 of its `Init()` method. This code was generated by the Window Editor.

The subform is created at line 99 of the `Init()` method of the `EditAdventureWindow`. The next two lines show the subform and set the selective relationship using the `#Adv_ID` field. This field relates the parent server (`AdvHdr`) to the child server (`AdvDtl`).

Manual Notification

The data server classes were designed to allow you to create multiple instances of the same server without having to worry about work areas, unique aliases and SQL cursors.

When you create individual instances of the server, it is important to remember that notification will be sent only to its registered clients. If it has no clients, no notification is sent.

Consider the case where you wish to validate a key for uniqueness. You could use the following code:

```
LOCAL cCustID
cCustID := "00001"
oCust := CUSTOMER{}
IF !oCust:Seek(cCustID)
    ? "Customer is unique!"
ELSE
    ? "Customer exists!"
ENDIF
```

Using this type of code is perfectly safe and no other considerations must be made.

However, consider the case where you have a customer window on the screen and you do not have your window registered with the oCust data server. Your program executes the following:

```
oCust := CUSTOMER{}
IF !oCust:Seek(cCustID)
    oCust:NAME := "NEW NAME"
ENDIF
```

Since the customer window is not a registered client of oCust, it will not be notified of the update made to its database.

If you want your customer window to be updated, you must send the notification yourself.

Broadcast Message Activation

In the South Seas Adventures application, this type of custom notification was accomplished using a notification broadcasting system from the SSAWindow class (the shell window of the application).

Here's how it works. Essentially, any code that modifies servers directly sends a notification to the SSAWindow. The message is sent via the BroadcastMessage() method of the SSAWindow class.

Broadcasting Messages

The BroadcastMessage() method is called from several different methods, including the Notify() method for edit windows and others containing custom code for push buttons (for example, Delete, Invoice, OK, Refund, and Void). You can find these methods by using the Source Code Editor's Find dialog box, and selecting the Advanced >> push button.

Open the Adventure:Methods module and double-click on the OKButton() method in the NewAdventureWindow class. Several key lines are shown below:

```
METHOD OKButton() CLASS NewAdventureWindow
...
IF ValidateControls(SELF, SELF:AControls)
...
    SELF:Append()
...
    oCust := Dup_Customer{}
    oCust:Seek(oDCmCustID:Value)
    SELF:Append()
...
    // Updates to Adventure servers
...
    SELF:Server:Commit()
...
    // Broadcast notification
    SELF:Owner:BroadcastMessage(SELF,;
        #Customer)
    SELF:Owner:BroadcastMessage(SELF,;
        #Adventure)
...
    SELF:EndWindow()
ENDIF
RETURN SELF
```

The message to be broadcast is simply the symbolic name of the affected server. The SSAWindow window, in turn, sends the notification to all its child windows that possess a ReceiveBroadcastMessage() method. The messages to be broadcast simply contain the symbolic name of the affected server:

```
METHOD BroadcastMessage(oSender, symMessage) ;
CLASS SSAWindow
LOCAL i AS WORD
LOCAL oCurrentChild AS OBJECT
FOR i := 1 TO LEN(aChildWindows)
    oCurrentChild := aChildWindows[i]
    // Do not process the sender
    IF oSender != oCurrentChild
        IF IsMethod(aChildWindows[i],;
            #ReceiveBroadcastMessage)
            oCurrentChild:;
                ReceiveBroadcastMessage;
                    (symMessage)
        ENDIF
    ENDIF
NEXT
RETURN SELF
```

Receiving
Broadcast Messages

In the South Seas Adventures application, several windows have a ReceiveBroadcastMessage() method. These include browse windows (Adventure, Employee, Item, Invoice, Payment), edit windows (Adventure, Employee, Item, and Invoice), and the NewAdventure and ViewPayment windows.

The child window that needs to receive these notifications can have its own ReceiveBroadcastMessage() method to update itself accordingly.

For example, open the Adventure:Methods module and double-click on the `ReceiveBroadcastMessage()` method for the AdventureBrowser window:

```
METHOD ReceiveBroadcastMessage(symMessage);
  CLASS AdventureBrowser
  IF symMessage == #Adventure
    oSFAdventureSubform:Browser:Refresh()
  ENDIF
RETURN NIL
```

In the previous example, the AdventureBrowser and the NewAdventureWindow windows each have their own instances of the Adventure data server. When a new adventure is added, the AdventureBrowser window must be notified.

Summary

You now know how to use the DB Server and SQL Server Editors to create data servers. In this lesson, you have:

- Created the Customer class (a subclass of DBServer), as well as the AccInv and AccPay classes (subclasses of SQLTable)
- Created an ODBC data source and learned how to program using data servers
- Been introduced to the notification process that occurs between data servers and data windows

For more information please refer to the *IDE User Guide*.

Defining Field Specifications

This lesson introduces you to field specification (or *field spec*) entities. You will learn how to use the FieldSpec Editor to define field specs. These definitions save you time when creating data servers with the DB Server Editor and creating data windows with the Window Editor. Defining reusable field spec entities also helps you ensure consistency in your database definitions.

Overview

In many cases, the different data servers in your application contain similar, if not identical, fields. For example, phone numbers—whether they are home, business, fax, or cellular numbers—are usually the same length. Other common examples are key fields, like customer, account, and employee codes, that might be used to relate your data in many files. You can either define the properties of these common fields each time you create a new data server or you can create a template, or field spec, that you reuse in each data server that needs it.

A field spec is essentially a set of properties (such as validation and formatting rules), that are related to a field, but are independent of a particular data server. Thus, when defining data servers, you can use the same property values for common fields, and any change made to a field specification is automatically propagated to all data servers that use that field spec.

Also, as you will see in [Chapter 5: Creating and Using Windows](#), a data form initially created with the Window Editor's Auto Layout feature, uses field definitions and properties specified in field specs in a data server.

Careful design of your field properties can save you time when later defining data servers and designing data-entry windows.

Exercise

In this exercise, you will create generic Phone and Customer ID field specs that you will use to describe fields in a data server. You will further refine the definitions in order to save time in future data server and window definitions.

Creating and Modifying Field Specifications

Invoking the FieldSpec Editor



The FieldSpec Editor allows you to create and modify field specifications. To invoke the FieldSpec Editor:

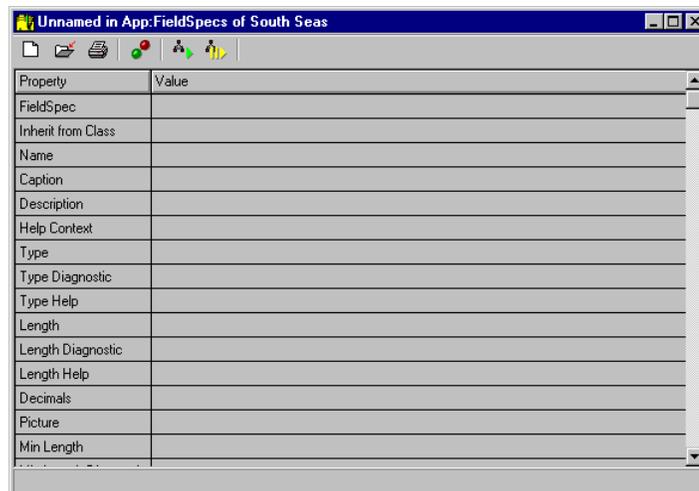
1. Open the South Seas application by double-clicking its branch on the Repository Explorer tree View.
2. Create a new module by clicking the New Module toolbar button.
3. In the resulting Create Module dialog box, type **App:FieldSpecs** and choose OK.

Select the App:FieldSpecs module.



5. Select the FieldSpec Editor command from the Tools menu (or click the Open Entity toolbar button and select FieldSpec Editor).

The FieldSpec Editor window displays:



Notice how similar this window is to the FieldSpec Properties window in the DB Server Editor. One difference is that there are no properties whose names have the FS prefix (for example, Name, Caption, Description, and Help Context) like in the DB Server Editor. The FieldSpec Editor also has its own toolbar and status bar.

Creating a Field Spec

To create a field spec entity, you must specify at least four properties: FieldSpec, Name, Type, and Length, as follows:

1. Select the FieldSpec property and type **PHONEFS**.
2. Press Enter.
PhoneFS is the name of the field specification. It must be unique among all field spec definitions.
3. Select the Name property and type **PHONE_FS**.
4. Press Enter.
Phone_FS is the symbolic name of the field specification. It must be unique in your application.
5. Select the Type property and choose Character from the drop-down list box.
The Length property is automatically set to 10, which is appropriate.
6. Save your field spec by clicking the Save toolbar button.
The basic minimal Phone field spec definition is complete.
7. Clear the FieldSpec Editor by choosing the New FieldSpec command from the File menu (or the corresponding toolbar button).
8. Repeat the previous steps for a CUST_IDFS field spec with CUST_ID_FS as the symbolic name. Set the Type to Character and specify a length of 5.
9. Close the FieldSpec Editor by double-clicking its system menu.



Planning Data Server Field Properties

In [Chapter 3: Working with Data Servers](#), you imported an existing .DBF file into the DB Server Editor. For each field of the .DBF file, a field spec entity and its associated properties were automatically defined. However, the auto layout feature fills in only the most basic properties, such as Name, Type, Size, and Caption. It creates these properties using field information from the database.

This is a powerful feature and a time saver for most of your fields, but some things cannot be determined by the DB Server Editor. For example, the Picture property for phone numbers or the fact that a Customer ID code is always required (because it is a key field) are not available from the DBF import process.

Also, because some field names are cryptic, the resulting generated properties may not be intuitive. So, let's plan ahead and modify several properties, including captions and messages, to make them more meaningful:

1. Open the PhoneFS field spec entity from by double-clicking in it in the list view of the Repository Explorer.

The FieldSpec Editor window displays.

2. Select the Picture property, type **@R (999)999-9999** and press Enter.
To offer field-level help to your users, specify the Help Context property as follows:



3. Select the Help Context property. Type **Phone_Numbers** and press Enter.
4. Save your new field spec by choosing the Save toolbar button.

5. Close the FieldSpec Editor by double-clicking its system menu.

Typically, key fields like Customer or Employee codes are required in order to relate one file to another. To do this, perform the following steps:

6. Open the Cust_IDFS field spec entity by double-clicking on it in the list view of the Repository Explorer.

7. Select the Required property, and select Yes from the drop-down list box.

Whenever the Cust_IDFS field spec is used and does not meet the above requirement, an error message displays indicating the failure. Make this message more informative by specifying your own message, as follows:

Select the Required Diagnostic property and type:

The Customer ID code is MANDATORY!

Note: When defining an application-wide field spec, be careful of descriptions, captions, and messages. Instead of using “Enter the employee’s phone number,” you should be more generic and type “Enter the phone number.”



9. Save your new field spec by choosing the Save toolbar button.

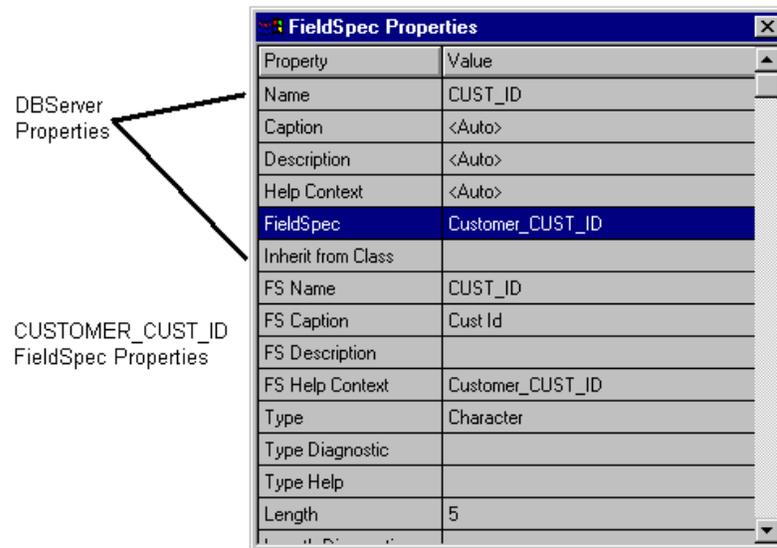
10. Close the FieldSpec Editor by double-clicking its system menu.

Attaching a Field Spec to a Data Server Field

Now, let’s redefine the data server field specifications using your newly defined field specs.

1. Open the Customer:Data module by clicking its branch on the Repository Explorer.
2. Open the DB Server Editor by double-clicking the Customer server entity on the Repository Explorer list view.
3. Select CUST_ID from the Fields Include list box.

The FieldSpec Properties window displays the field spec properties for this field:



Let's apply your field spec definitions to the Customer data server's Cust_ID field, using the following procedure:

1. In the FieldSpec Properties window, select the FieldSpec property. From the drop-down list box, select CUST_IDFS.
2. Click the Caption property and type **Cust ID:**.
3. Scroll down through the FieldSpec Properties window to see the Required and Required Diagnostic properties, have automatically been filled in.

As stated earlier, field spec entities are independent of the data server. You can use any field spec which is currently defined in your application. In this case, the PhoneFS field spec entity has most of what we need for both the Phone and Fax phone number field.

4. From the Include list box in the DB Server Editor, select the Phone field.
5. In the FieldSpec Properties, select the FieldSpec property. From the drop-down list box, select PHONEFS.
6. Click the Caption property and type **Phone:**.
7. Scroll down through the FieldSpec Properties window to locate the Picture property.

The picture clause from the PhoneFS fieldspec displays.

8. Repeat the steps 4 through 7 for the Fax field, applying the PhoneFS FieldSpec and setting the caption to Fax:.
9. Save your new data server definition by choosing the Save toolbar button.

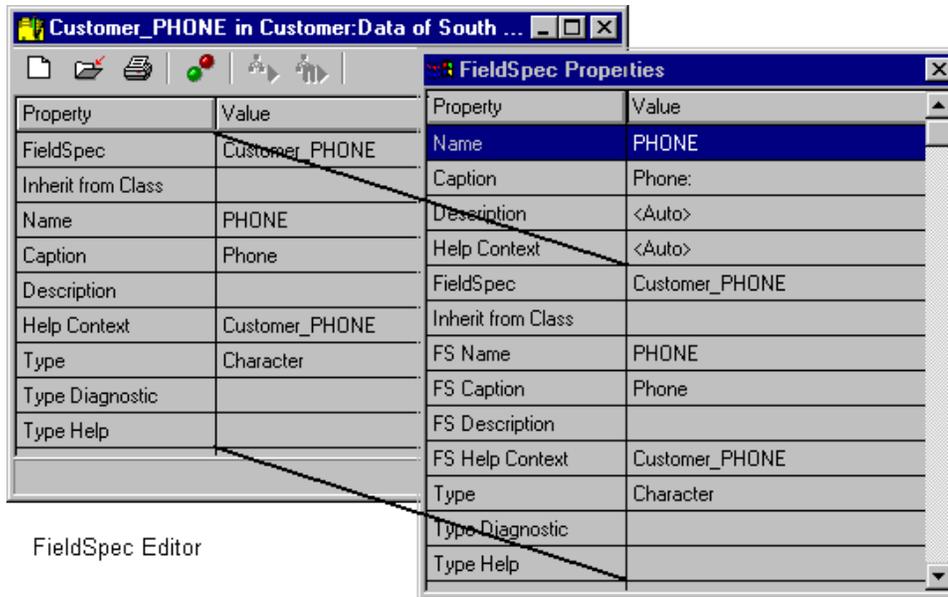


Creating Field Specs from the DB Server Editor

It is important to realize that the field properties shown within the DB Server Editor include five properties that relate to the field itself (Name, Caption, Description, Help Context, and FieldSpec). The remaining properties belong to the specified field spec entity.

The seventh to tenth properties have names that begin with “FS” to emphasize properties of the field spec.

Any modification of the properties from the FieldSpec property line to the bottom of the list affects the field spec entity directly. You can define both data server and common field specs using the same tool:



FieldSpec Editor

DB Server Editor
FieldSpec Properties window

Although this lesson has focused on the importance of the reusability of field specs, many field specs are unique to a given database. For these, the DB Server Editor offers the convenience of having access to all of the fields and their attached field spec in one place.

Let's close the DB Server Editor (by double-clicking its system menu) and review the benefits of defining reusable field specs as follows:

- A field spec is required to define each field of every data server. Each field spec contains more than 20 properties usable by the data server. As you will see in the “Creating and Using Windows” chapter, data windows make use of data servers' field spec properties. Note that three of those properties (Caption, Description, and Help Context) are hierarchical in nature and may be redefined for use by the window.
- Whether you are prototyping or in production, the earlier in the sequence you define the field specs, the earlier you'll gain consistency and reusability.

For example, the South Seas Adventures application uses the PhoneFS FieldSpec for the Phone and Fax fields in the Customer data server. This server is used in three data windows (new, edit, and subform customer windows). The choice is clear—you can define a property (like Picture) once, in a field spec used by both fields, twice for the two DB server fields, or six times in different windows. Defining it once saves time, and makes it easier to make changes in the future.

Summary

In this lesson, you have created reusable field spec definitions using the FieldSpec Editor. You have replaced data server-specific field specs with generic field specs. Additionally, you have learned about the relationship between field specs, data server field specifications, and field spec entities.

Creating and Using Windows

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, we will discuss other window types available within CA-Visual Objects 2.7. These windows include, Dialog, DataDialog, Data, and Shell Forms.

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. A 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.

Top Application Windows

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.

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 2.7 desktop is a good example of an MDI application:

- Each child window has a frame, system menu, Maximize, Minimize and Close buttons, and an icon.
- The user can control the Child window 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 Forms

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 Forms

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 2.7 allows both modal and modeless dialog windows.

Modal Dialog Forms

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.

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.

DataDialog Forms

A DataDialog window is a window that combines features from both data windows and dialog windows. This combination allows the creation of modal, data-aware windows.

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.

Data Forms

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 assigned to it. 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.

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 [Chapter 4: Defining Field Specifications](#) in this guide.) If data fails the validation test, the diagnostic message of the validation rule displays 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.

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 displays very quickly and the methods of the window have enough information to act intelligently.

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—the record append occurs.

Exercise

Viewing a MDI Application

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

The Shell Form

The South Seas Adventures application uses a shell window as its MDI parent window. It was created using the Window Editor and can be inspected by double-clicking on the SSAWindow binary entity of the SSA Shell:Forms module in the Repository Explorer list view.

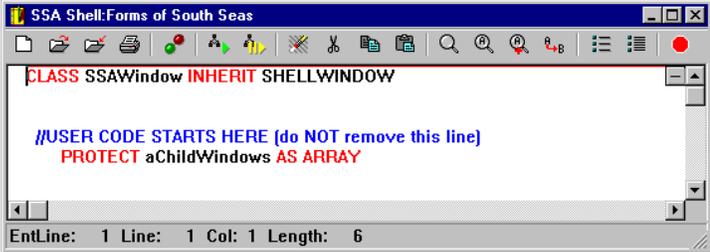
Adding Functionality

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. Fortunately, there are two methods that can be used to enhance the code that the Window Editor generates for you. These are the PreInit and PostInit methods which are automatically called by the Init method of each class.

In the South Seas Adventures application, there is a need to keep track of all child windows that have been instantiated. Let's see what is involved:

1. Open the SSAWindow class by double-clicking on it in the Repository Explorer list view.

You are presented with a Source Code Editor window:



```
CLASS SSAWindow INHERIT SHELLWINDOW

//USER CODE STARTS HERE (do NOT remove this line)
PROTECT aChildWindows AS ARRAY
```

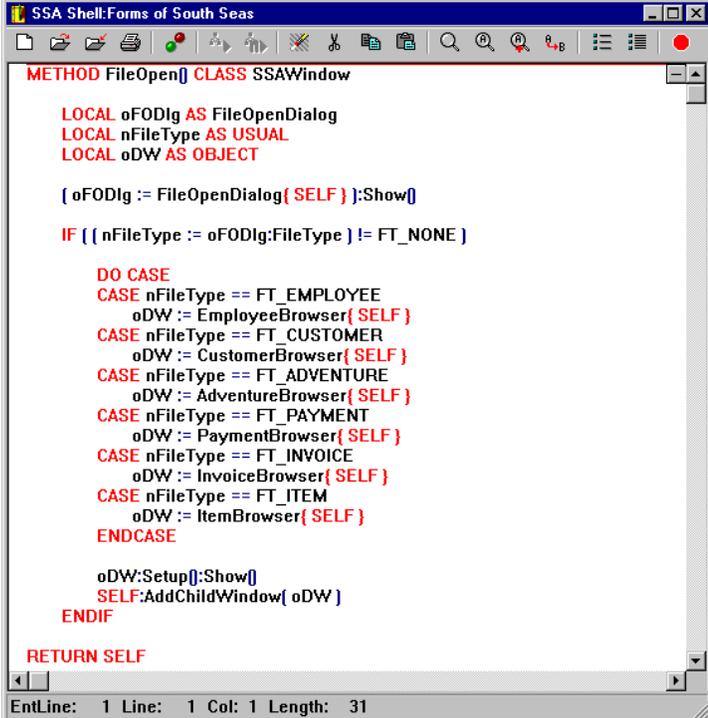
EntLine: 1 Line: 1 Col: 1 Length: 6

The window shows that a protected variable called `aChildWindows` has been added to the `SSAWindow` class, which was generated from the Window Editor. A reference to each child window is stored in the `aChildWindows` array as it is opened.

2. Close the Source Code Editor by double-clicking its system menu.
3. Let's see how child windows are opened and tracked.

Double-click the `SSAWindow:FileOpen()` method in the Repository Explorer list view.

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



```

METHOD FileOpen() CLASS SSAWindow
    LOCAL oFODlg AS FileOpenDialog
    LOCAL nFileType AS USUAL
    LOCAL oDW AS OBJECT

    { oFODlg := FileOpenDialog( SELF ) }:Show()

    IF { ( nFileType := oFODlg:FileType ) != FT_NONE }

        DO CASE
        CASE nFileType == FT_EMPLOYEE
            oDW := EmployeeBrowser( SELF )
        CASE nFileType == FT_CUSTOMER
            oDW := CustomerBrowser( SELF )
        CASE nFileType == FT_ADVENTURE
            oDW := AdventureBrowser( SELF )
        CASE nFileType == FT_PAYMENT
            oDW := PaymentBrowser( SELF )
        CASE nFileType == FT_INVOICE
            oDW := InvoiceBrowser( SELF )
        CASE nFileType == FT_ITEM
            oDW := ItemBrowser( SELF )
        ENDCASE

        oDW:Setup():Show()
        SELF:AddChildWindow( oDW )
    ENDIF

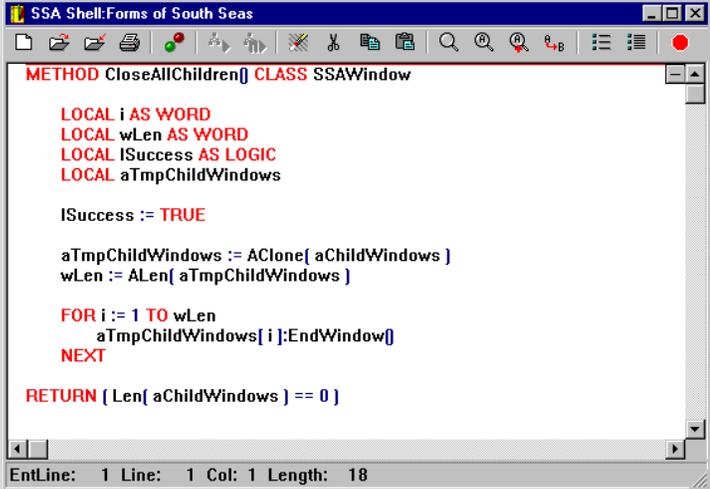
    RETURN SELF
  
```

EntLine: 1 Line: 1 Col: 1 Length: 31

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.

4. Close the Source Code Editor and double-click the `SSAWindow:CloseAllChildren()` method.

The code for the CloseAllChildren() method displays:



```
METHOD CloseAllChildren() CLASS SSAWindow
LOCAL i AS WORD
LOCAL wLen AS WORD
LOCAL ISuccess AS LOGIC
LOCAL aTmpChildWindows

ISuccess := TRUE

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

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

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

EntLine: 1 Line: 1 Col: 1 Length: 18

5. When you are finished examining the source code, close the Source Code Editor window by double-clicking its system menu.

Creating a Modal Dialog Box

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

Warning Box Modal Dialog Forms

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. Select the SSA Shell:Forms module on the Repository Explorer tree view, and double-click the SSAWindow:QueryClose() method in the List view.

The Source Code Editor appears:

```

METHOD QueryClose( oEvent ) CLASS SSAWindow

LOCAL oWB AS WarningBox
LOCAL ILeave := 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
  ILeave := TRUE
ENDIF

RETURN ILeave

```

Line 9 creates an instance of the WarningBox class and stores it in the oWB variable. Line 11 then assigns the Icon type and the Button configuration to the class. Finally, the class is shown and the return value is tested against the constant to see if the reply was yes.

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 its system menu.

Retrieving Values from Modal Dialog Forms

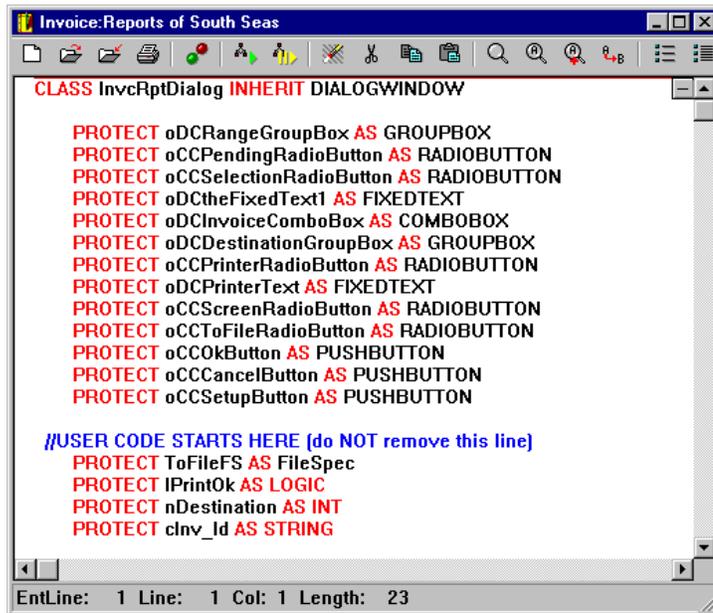
It is pretty simple to get one result back from a dialog box, but how about many results?

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(), the Window is not destroyed. This means that the control objects can be queried after the user clicks the OK or Cancel buttons. Unfortunately they can not be interrogated from outside of the class because there are no ACCESS methods written for a DialogWindow object.

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

1. Use the Window Editor to create a dialog form.
2. Create any necessary instance variables:



```

CLASS InvcrptDialog INHERIT DIALOGWINDOW

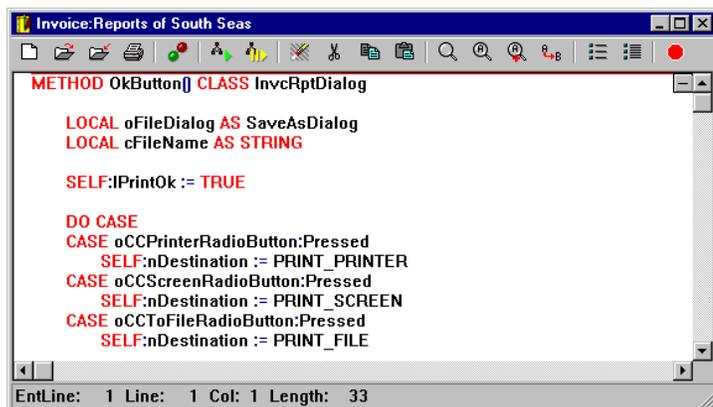
PROTECT oDCRangeGroupBox AS GROUPBOX
PROTECT oCCPendingRadioButton AS RADIOBUTTON
PROTECT oCCSelectionRadioButton AS RADIOBUTTON
PROTECT oDCtheFixedText1 AS FIXEDTEXT
PROTECT oDCInvoiceComboBox AS COMBOBOX
PROTECT oDCDestinationGroupBox AS GROUPBOX
PROTECT oCCPrinterRadioButton AS RADIOBUTTON
PROTECT oDCPrinterText AS FIXEDTEXT
PROTECT oCCScreenRadioButton AS RADIOBUTTON
PROTECT oCCToFileRadioButton AS RADIOBUTTON
PROTECT oCCOkButton AS PUSHBUTTON
PROTECT oCCCancelButton AS PUSHBUTTON
PROTECT oCCSetupButton AS PUSHBUTTON

//USER CODE STARTS HERE [do NOT remove this line]
PROTECT ToFileFS AS FileSpec
PROTECT IPrintOk AS LOGIC
PROTECT nDestination AS INT
PROTECT clnv_Id AS STRING

```

EntLine: 1 Line: 1 Col: 1 Length: 23

3. Update the instance variables when the user clicks OK:



```

METHOD OkButton() CLASS InvcrptDialog

LOCAL oFileDialog AS SaveAsDialog
LOCAL cFileName AS STRING

SELF:IPrintOk := TRUE

DO CASE
CASE oCCPrinterRadioButton:Pressed
SELF:nDestination := PRINT_PRINTER
CASE oCCScreenRadioButton:Pressed
SELF:nDestination := PRINT_SCREEN
CASE oCCToFileRadioButton:Pressed
SELF:nDestination := PRINT_FILE

```

EntLine: 1 Line: 1 Col: 1 Length: 33

Create the ACCESS and ASSIGN methods to allow us access from outside of the class:

```

ACCESS clnv_Id CLASS InvcrptDialog
RETURN SELF:clnv_Id

ASSIGN clnv_Id( uValue ) CLASS InvcrptDialog
SELF:clnv_Id := uValue
RETURN SELF:clnv_Id

ACCESS IPrintOk CLASS InvcrptDialog
RETURN SELF:IPrintOk

ASSIGN IPrintOk( uValue ) CLASS InvcrptDialog
SELF:IPrintOk := uValue
RETURN SELF:IPrintOk

ACCESS nDestination CLASS InvcrptDialog
RETURN SELF:nDestination

ASSIGN nDestination( uValue ) CLASS InvcrptDialog
SELF:nDestination := uValue
RETURN SELF:nDestination

ACCESS ToFileFS CLASS InvcrptDialog
RETURN SELF:ToFileFS

ASSIGN ToFileFS( uValue ) CLASS InvcrptDialog
SELF:ToFileFS := uValue
RETURN SELF:ToFileFS

```

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 Form

In this exercise, you are going to create a data form to support the editing of customer records. This data form 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 clicking its branch on the Repository Explorer tree view.
2. Select the Import command from the File menu.
3. From the Import dialog box, select the Files of Type ComboBox and select Mod. Import Files (*.mef). After choosing the .MEF files, select the TUTWIND.MEF file located in the CA-Visual Objects 2.7 SAMPLES\SSATUTOR\FILES subdirectory.
4. Choose OK.

A new module, Tutorial:Windows has been added to the application.

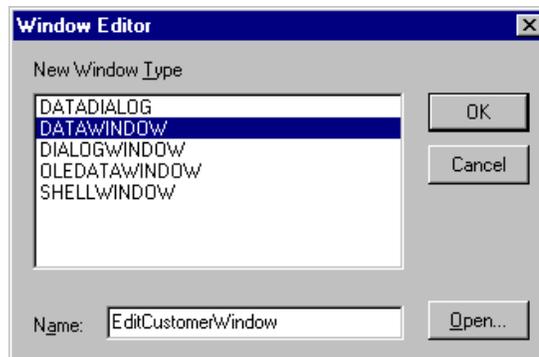
Creating a Data Window Template

To create a data window template:

1. Select the Customer:Forms module by clicking its branch on the Repository Explorer tree view.
2. Select the Window Editor command from the Tools menu or select the New Entity toolbar button.

The Window Editor dialog box appears, allowing you to define the type of window.

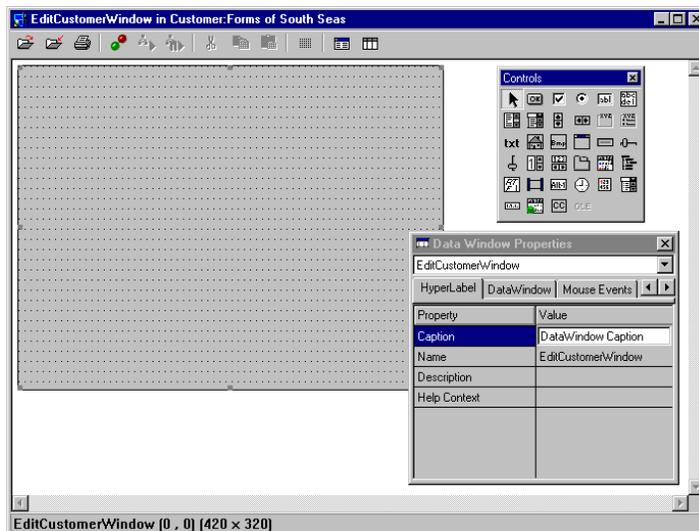
3. Select DATAWINDOW:



4. Type **EditCustomerWindow** in the Name edit control.

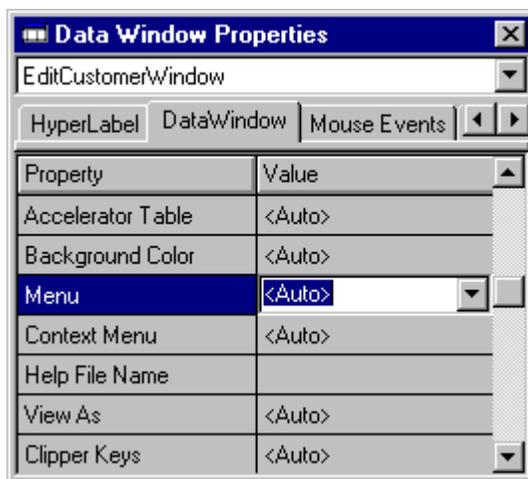
- Choose OK.

The Window Editor displays:



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

- In the Data Window Properties box, choose the Menu property (found under the DataWindow tab) and select SSACHILDMENU in the drop-down list box:



This attaches the SSACHildMenu menu to the data form.

- Choose the Caption property (found under the Hyperlabel tab) and type **Edit Customer**.

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

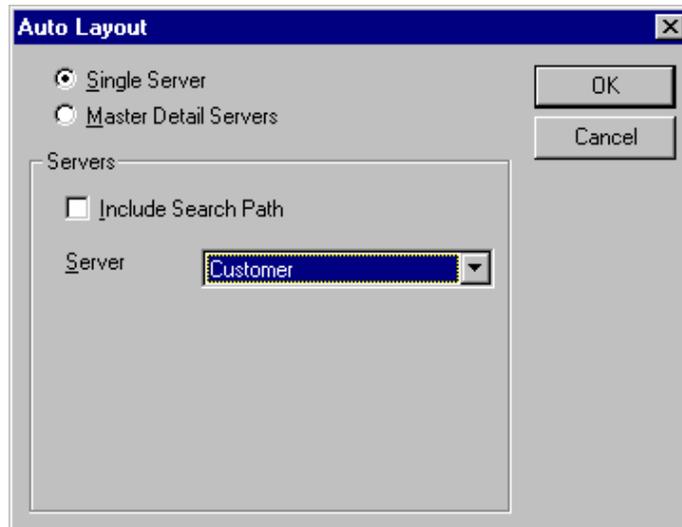
Designing Your Window Layout

Auto Layout provides 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:



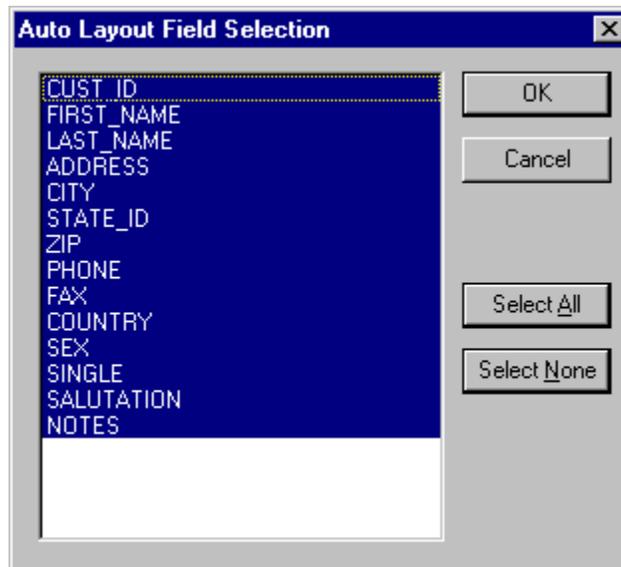
1. Click the Auto Layout toolbar button.

The Auto Layout dialog box appears:



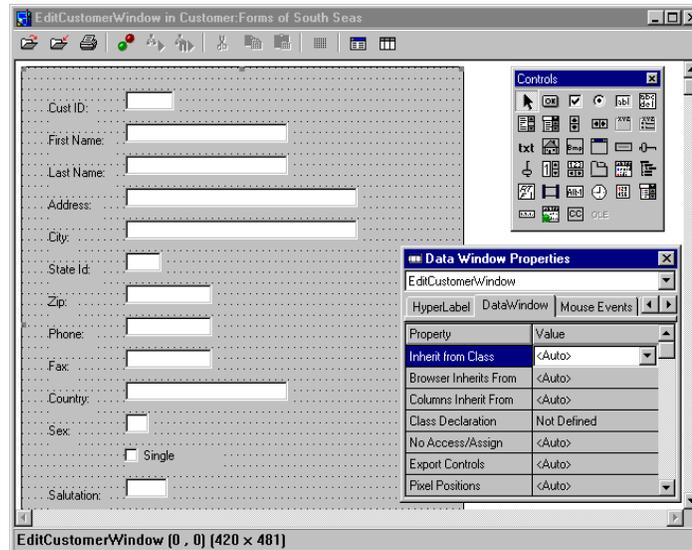
2. Select the Single Server option.
3. Select CUSTOMER from the Server drop-down list box.

Choose OK.



- Choose OK to select all of the fields for auto-layout.

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



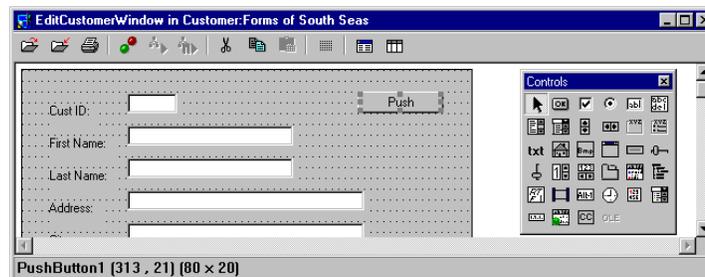
If you want, you can edit, move, or delete any of these fixed text and single-line edit controls from the predefined data form.

Adding a Push Button

Now let's add a push button to the data form:

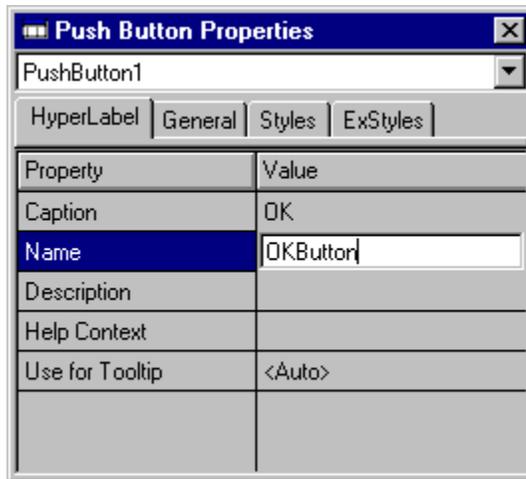


- Select the Push Button icon from the Window Editor's tool palette.
- Drop the push button control on the top right-hand corner of the data form template:



- Change the Caption property (found under the Hyperlabel tab of the Push Button properties) to **OK**.

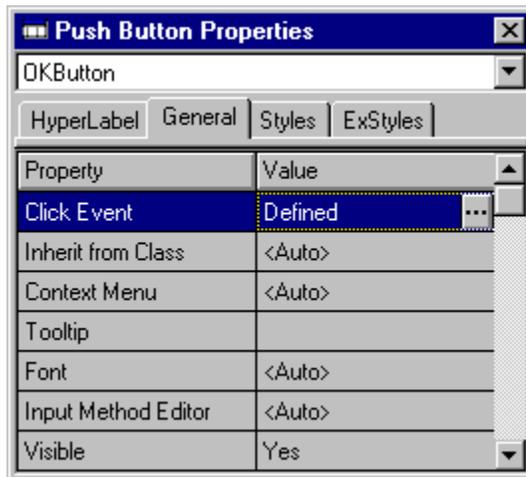
- Change the Name property to **OKButton** (found under the Hyperlabel tab of the Push Button properties):



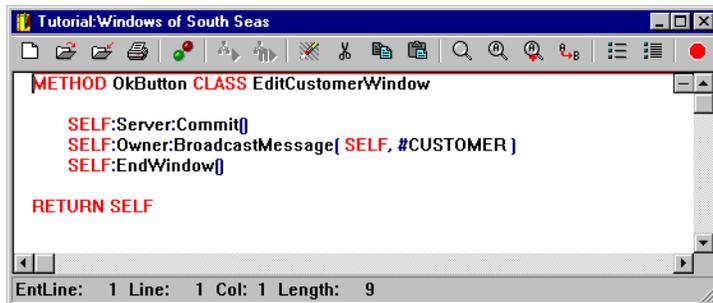
This conforms to the naming convention used throughout the application.

- Select the Click Event property (found under the General tab of the Push Button Properties) and click 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:



```

METHOD OkButton CLASS EditCustomerWindow
    SELF:Server:Commit()
    SELF:Owner:BroadcastMessage( SELF, #CUSTOMER )
    SELF:EndWindow()

RETURN SELF
  
```

EntLine: 1 Line: 1 Col: 1 Length: 9

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 its system menu.
2. Close the Window Editor by double-clicking its system menu.
3. Select Yes when prompted to save the changed entities.
4. Build the application by clicking the Build toolbar button.
5. Run the South Seas Adventures application by clicking the Execute toolbar button.
6. Log in as usual (Name: **User**, Password: **Trainee**).
7. Select the Open command from the File menu.
You are presented with the Open File dialog box.
8. Start a Customer file edit session by clicking the Customer radio button.
9. Choose OK.
This opens the Customer Browser window.
10. Click the name Baker in the Customer Browser window, and then click the Edit toolbar button to open the newly created Edit Customer window.
11. Choose OK to close the Edit Customer window.
12. When you are finished, exit the South Seas Adventures application by double-clicking its system menu.

In [Chapter 6: Adding Controls to Your Windows](#) 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 windows. You should now understand the difference between SDI and MDI applications and have a good understanding of the most commonly used window types.

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.

Adding Controls to Your Windows

This lesson examines the many controls you can add to your windows. CA-Visual Objects 2.7 provides several predefined classes that your program can use to create these controls. When you finish this lesson, you should be familiar with all of the controls on the Window Editor Tool Palette.

Overview

Controls allow users to communicate with an application. They can be placed on data, dialog, and data dialog windows. The difference between these windows is that on Dialog windows, the data associated with a control is buffered by the control, with data windows or data dialogs, controls can be tied directly to a field in a data server. Also, some controls are tied specifically to a data window (like a subdata window control).

You may have wondered how the data window in the previous lesson is capable of storing and retrieving data between the database and the controls on the Customer window. Essentially, the data window uses name-based linkages to the data server.

If a control name has a corresponding field in the data server, the data window automatically associates the two when retrieving and storing data. If a control has no corresponding field in a data server, the data held in the control is not automatically retrieved from or stored to a data file, and consequently, must be manipulated by your program.

When you created the Customer window using Auto-Layout, single-line edit controls were created based on the Customer data server. Each data control is named according to the fields in the data server.

Exercise

In the previous lesson, you briefly saw single-line edit controls. The following exercise provides greater detail about the controls listed below:

- Single-line Edit
- Multiline Edit
- Combo Box
- Check Box
- Radio Button
- Radio Button Group
- List Box
- Group Box
- Fixed Icon

All of these controls can be created using the Window Editor Tool Palette.

Single-line Edit (SLE) Controls

Single-line edit controls are the most widely used type of data-entry control. These controls are ideally suited to fields such as names, descriptions, numeric amounts, and dates.

Data displays in the control as text. The user can enter and delete characters, as well as cut, copy, and paste text. The characters in the control are formatted according to the field specification of its corresponding field in the database, or according to an attached FieldSpec object.

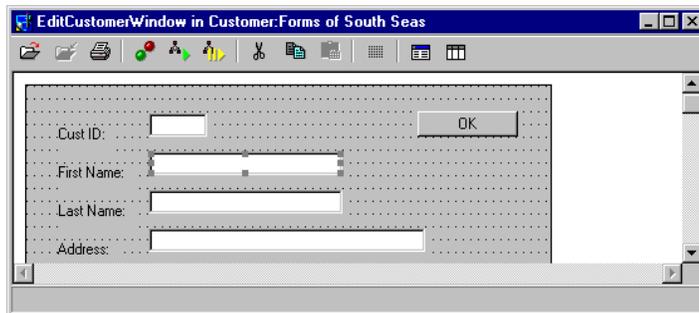
Single-line edit controls can be used to capture text, numeric, date, and logical types of data. Any necessary conversions are done automatically by the data window.

Single-line Edit
Controls as Data
Servers

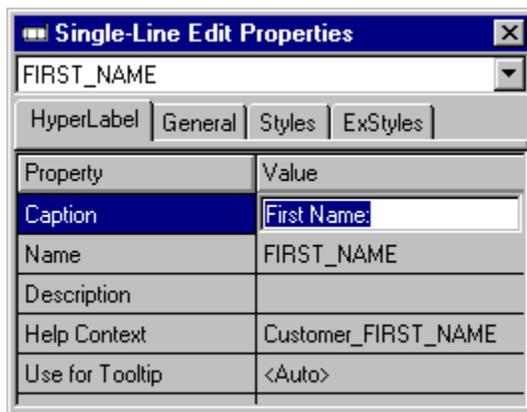
The following exercise demonstrates the use of single-line edit controls as data servers:

1. Open the Customer:Forms module by clicking its branch in the Repository Explorer tree view.
2. Open the EditCustomerWindow window entity by double-clicking it in the Repository Explorer list view.

3. Select the single-line edit control to the right of the First Name: label by clicking it:



4. Inspect the Name property in the Single-line Edit Properties window (under the HyperLabel tab). This represents the name of the field in the Customer data server (in this case, First_Name):



Multiline Edit (MLE) Controls

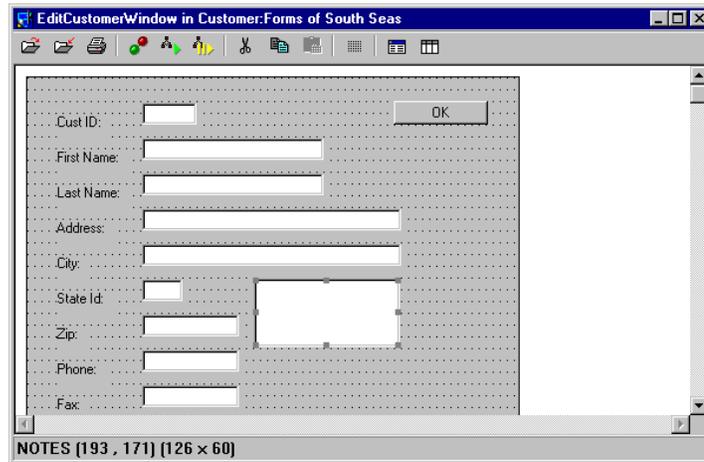
A multiline edit control differs from a single-line edit control in its ability to accept multiple lines of text. Editing functions (like cut and paste) are also available within this control. Multiline edit controls are suited to descriptive information, such as comments, notes, and addresses.

Moving the MLE Control

Now, let's move the Notes multiline edit control on the EditCustomerWindow. The Notes field is a memo field, and therefore lends itself well to a multiline edit control:

Note: CA-Visual Objects 2.7 will automatically generate a multiline edit control when using auto-layout on a memo field.

1. First we will delete the Notes: label next to the Notes multiline edit control. To do so, click the Notes: fixed text and press the Delete key.
2. Select the Notes multiline edit control by clicking it. Position it to the right of the State single-line edit control, about halfway across the window canvas area. Size it so that it is about 1 inch high and 2 inches wide:



Viewing Your Results

You can ensure that your changes have been made to the windows easily with CA-Visual Objects 2.7. The CA-Visual Objects 2.7 Window Editor allows fast prototyping with the use of the Window Editor Test Mode:

1. Select the View Menu.
2. Select Test Mode.

Your window now appears on top of the Window Editor displaying your changes.

3. To test that your controls are functional, click the Notes multiline edit control and type some text. Notice that the text that you have typed appears in the multiline edit control.
4. To return to the window editor, close the Test Mode window by clicking the close icon in the upper-right corner.
5. Click the Save Icon to save the changes.

Combo Box Controls

Using combo box controls, you can avoid the need for a user to know the exact data your program expects, as well as having to write validation code. For example, if the State_ID control were to remain a single-line edit control, it would require the user to be familiar with the abbreviations for 50 U.S. states. Additionally, every entry would have to be validated to insure that a valid code was entered.

We will now create a combo box control for the State_ID field:

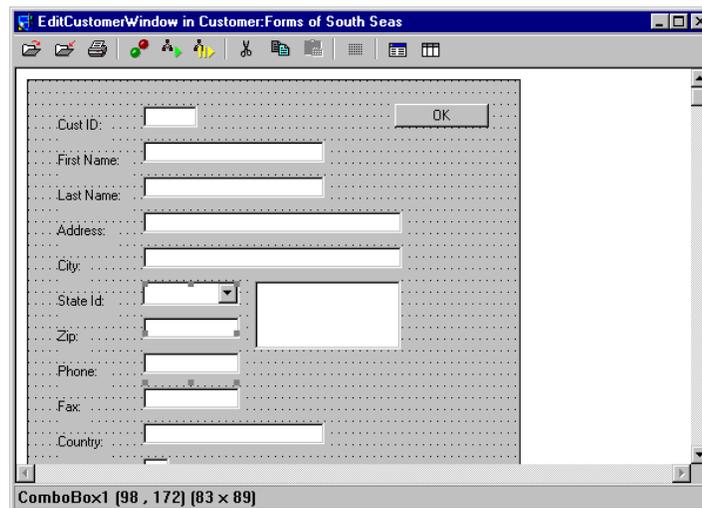
1. Delete the State_ID single-line edit control by clicking it and pressing the Delete key.



2. Click the combo box Tool Palette button.

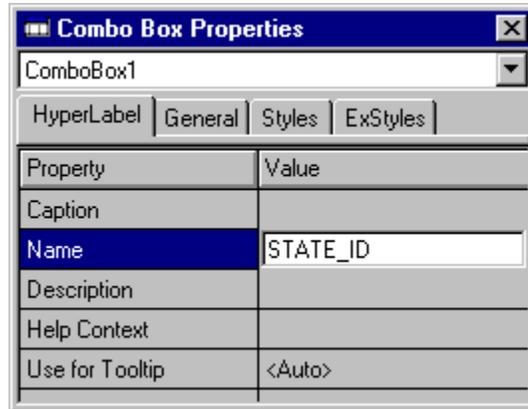
This selects a combo box control.

3. Drop the control by clicking the window canvas area. Position it next to the State ID: label. Size it roughly as shown in the next figure:

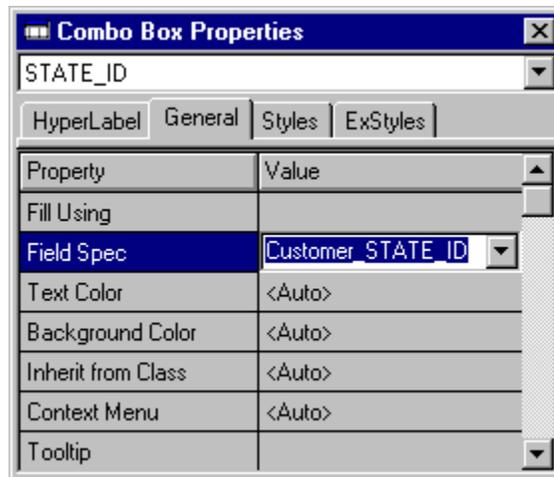


This control has not been tied to a field in any table yet. In the Properties window (under the HyperLabel tab), notice it is named ComboBox1.

- Change the Name property to **STATE_ID**. This ensures that the data window stores and retrieves values to and from the State_ID field of the Customer data server:



- You also want it to inherit specific field spec properties. In the Properties window, select the General tab. Select the FieldSpec property and change it by selecting CUSTOMER_STATE_ID from the drop-down list box:



This ensures that any special validations applied to the field specification are tied to your combo box.

- Now, you need to tell the combo box what to display. Select the Fill Using property, then click the ellipsis button.

You are prompted by the Fill Using dialog box. The Fill Using property of the combo box allows you to fill its list with either an array, the contents of a data server, or a method.

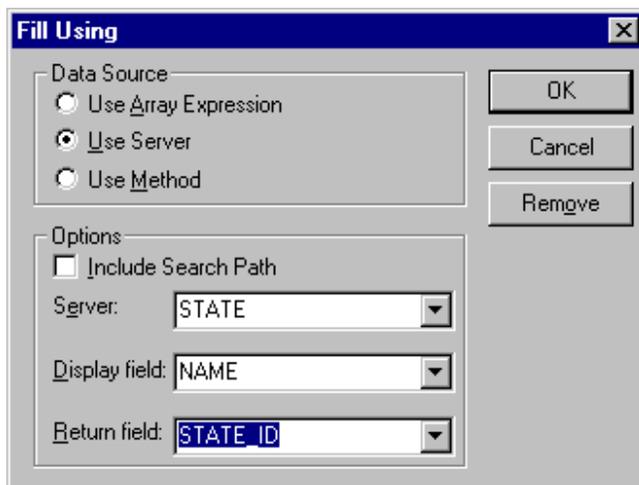
The State table and server have already been created and populated with valid state codes and names, so let's use it here.

- Select the Use Server radio button.
- From the Server combo box, select STATE.

The Server group box also provides the option to display one server field and return another to the field attached to the control. This feature allows you to retain your normalized databases while showing the user more descriptive information.

In this case, let's display the state name instead of the state code.

9. Select NAME from the Display field combo box.
10. Select STATE_ID in the Return field combo box, since it is the only field that appears in both servers:

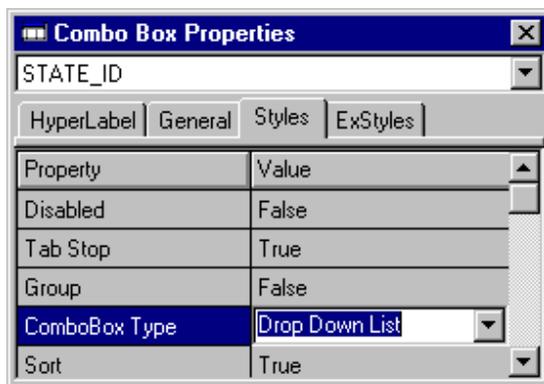


11. Choose OK to close the Fill Using Dialog box.

Invalid Code Entry Disabled

To disable a user's ability to enter invalid codes, you can change the style of the combo box to Drop-down List. Using this type of combo box forces the user to select a value in the list:

1. Select the Styles tab from the State_ID Combo Box properties box.
2. Select the ComboBox Type style and select Drop Down List from the drop-down list box:



Note: As seen previously, you can see your changes by running Test Mode.

Check Box Controls

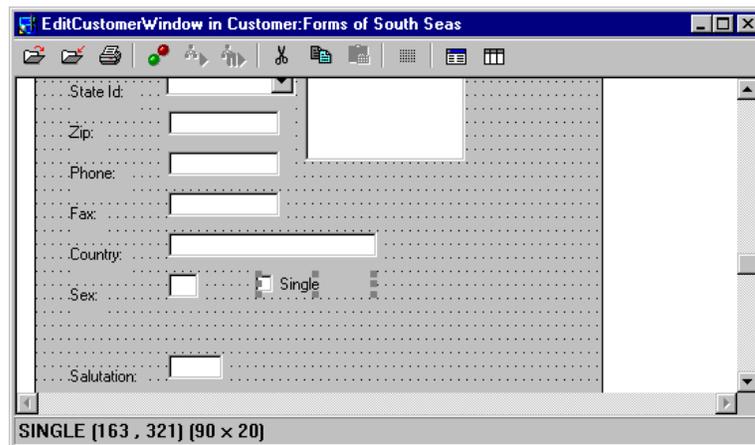
A check box is a square box with associated text that usually appears to the right of the check box. It acts as a toggle switch, allowing a user to turn an option on or off. Thus, it is usually used to represent logical fields.

When the check box is linked to a logical field in the data server, a value of TRUE in the server represents the checked (or ON) state of the check box, while a value of FALSE represents the unchecked (or OFF) state.

One of the properties of a check box provides for a three-state check box. The third state is dimmed and indicates that the check box status is unknown (or undefined).

The Customer data server has a logical field named “Single” to represent marital status. A check box control is automatically created by CA-Visual Objects 2.7 when using the Window Editor Auto Layout to represent a logical value in the database.

Let’s move the control by clicking the Single check box and position it next to the Sex: label:



Radio Button and Radio Button Group Controls

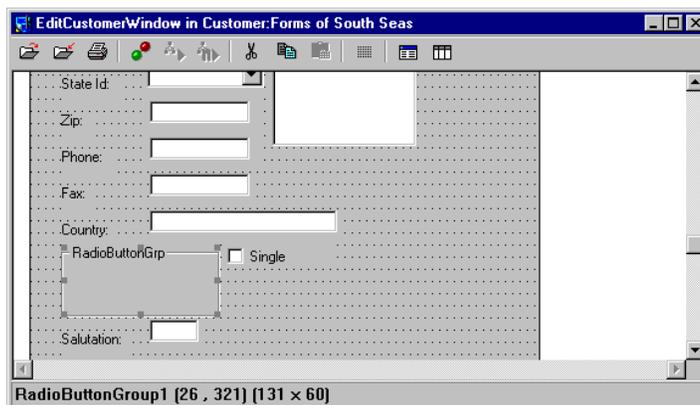
In radio button group controls, individual radio buttons provide mutually exclusive responses to a condition where only one choice is required. When you click a radio button, it is checked (ON). If you then click another radio button within the same radio button group, the radio button you first clicked on is unchecked (OFF).

The radio button is an oddity in the Button class, since it is not tied directly to a data field. Instead, it is grouped (with other radio buttons) into a radio button group control. Unless you create distinct radio button groups, all the radio buttons on a window are members of the same radio button group. A radio button group control can be tied to an actual field in a data server.

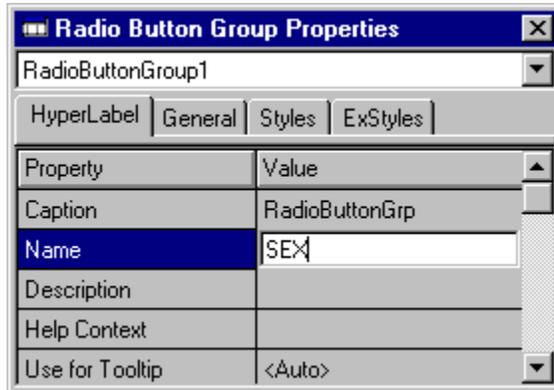
In the Window Editor, each radio button can be assigned a group value. When retrieving data from the server, the radio button group selects the radio button whose group value corresponds to the data value in the server. When storing data to the server, the radio button group uses the group value of the currently selected radio button.

You will now attach a radio button group, with Male and Female options, to the Sex field on the Customer window. The Sex field is an ideal candidate for radio buttons, since there are only two choices, Male and Female:

1. Select and delete both the Sex: label and single-line edit control.
2. Select the radio button group Tool Palette button and then move the mouse to the desired location on the window canvas area. Click the mouse button again to place and position the radio button group box.
3. Size the radio button group so that it is similar to the following:



- The radio button group is the control that gets linked to the data server. In the Properties window (under the HyperLabel tab), change the Name property to **Sex**:



- Change the Caption property to **&Sex**.

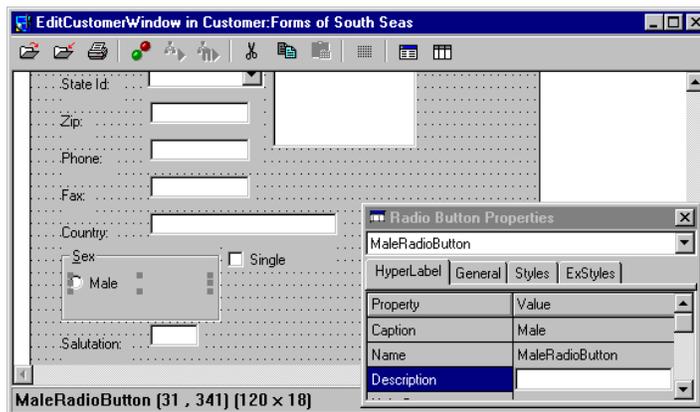
Notice that the S is now underlined. This allows the user to press Alt+S to move directly to the control. Now, let's place two radio buttons inside the radio button group box. The radio buttons can be placed in a column—one above the other.



- Click the radio button Tool Palette button and place a radio button in the radio button group box.

This radio button will be used for the Male option. The control is not directly linked to the data server, therefore its name should differ from the fields of the database. Although, it is not necessary to rename the field, it is still a good idea to do so.

- Now, click the Caption property and type **Male**. This displays on the window to the right of the radio button it represents.
- Type **MaleRadioButton** in the Name property:

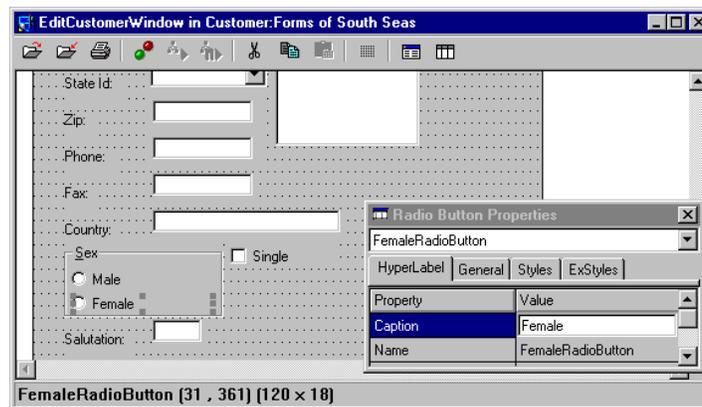


- Select the General tab on Radio Button Properties.

10. The Group Value property contains the actual value that is used by the radio button group. Click on this property and type **M**:



11. Add the Female radio button following steps 6 through 10. Place it below the Male radio button. You can set the Caption to **Female**, the name to **FemaleRadioButton**, and the Group Value property to **F**:



Note: As seen previously, you can see your changes by running Test Mode.

List Box Controls

The list box control is a collection of text strings. It displays as a scrollable, columnar list within a rectangle. A list box can allow either a single selection or multiple selections.

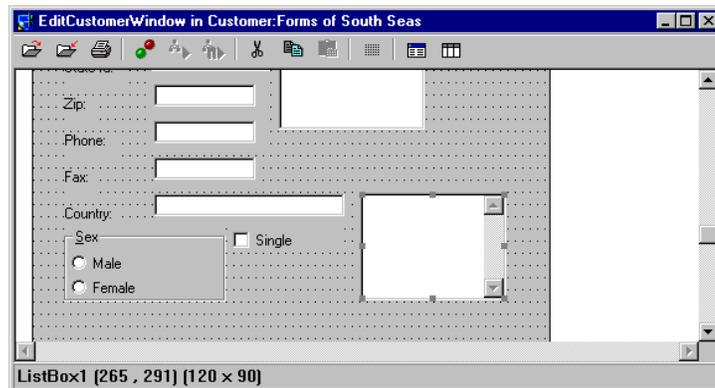
In a single selection list box, the user can select the item that the cursor is on by pressing the spacebar or clicking the left mouse button. In a multiple selection list box, the spacebar or mouse button toggles the selected state.

Navigation is accomplished by use of the vertical scroll bar and the navigation keys (Up/Down arrow and PageUp/PageDown keys) if there are more elements than can fit in the display area. Pressing a letter key moves the cursor and the selection highlight bar to the first item in the list starting with that letter.

You will use a list box to represent the salutation field of the customer data server. The list box has to represent the Mr., Mrs., and Ms. salutations.

You could have used a combo box or a radio button group to represent this, just as easily. Because you have already seen those controls, let's use the list box control here:

1. Delete the Salutation single-line edit control and label by clicking on each and pressing the Delete key.
2. Select the list box Tool Palette button. This selects a list box control.
3. Place the control by clicking the window canvas area. Position it to the right of the Country field as shown in the following figure. Size it to about 1.5 inches high by 1 inch:



This control has not yet been linked to a field in a table. In the Properties window, notice it is named **ListBox1**.

4. Change the Name property (under the HyperLabel tab) to **Salutation**.
This ensures that the data window stores and retrieves values to and from the Salutation field of the Customer data server.
5. Select the General tab in the List Box Properties.
6. You also want it to inherit specific field spec properties. In the Properties window, change the FieldSpec property to **CUSTOMER_SALUTATION**.

This ensures that any special validations applied to the field specification are always tied to your list box.

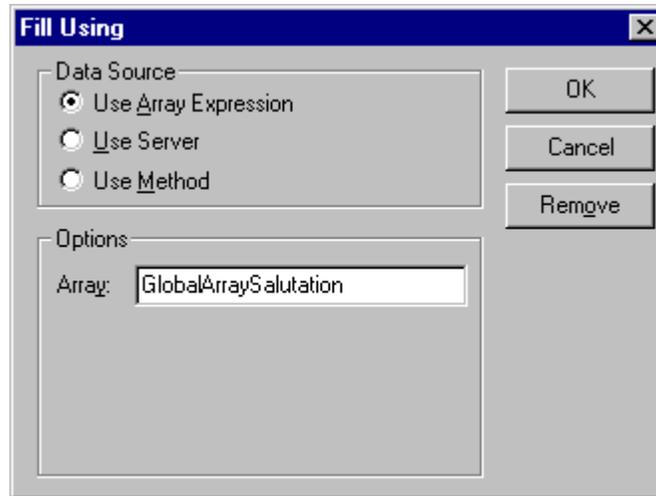
Now, we must create an array for use in filling the list box.

7. Select Repository Explorer from the Window menu to return to the Repository Explorer.
8. Select the App:Start() module and right-click.
9. Select Edit All Source in Module.
10. Notice the GlobalArraySalutation GLOBAL definition:

```
GLOBAL GlobalArraySalutation := {"Mr.", "Mrs.",  
"Ms."} AS ARRAY
```



11. Close the Source Code Editor by double-clicking its system menu.
12. Return to the Window Editor by selecting its session from the Window menu.
13. Now we need to tell the list box what to display. Select the Fill Using property (under the General tab) and then click the ellipsis button.
14. Select the Use Array Expression radio button is the default. Type **GlobalArraySalutation** in the Name edit control. Then choose OK:



Let's look at the available styles for the list box.

15. Click the Styles tab on the List Box properties.
16. Set the Vertical Scroll Bar property to **False**. You do not need the scroll bar because all of the items fit into the display area.

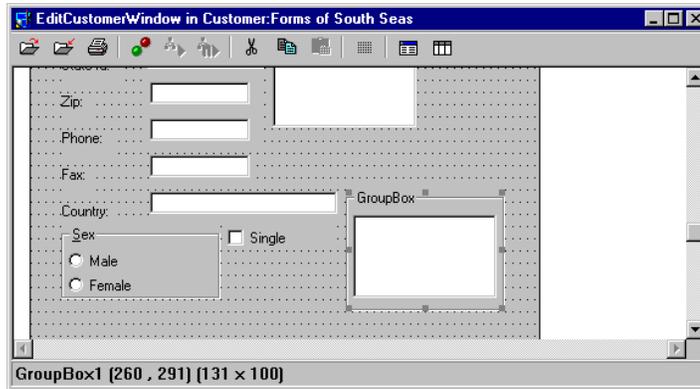
Group Box Controls

The group box control has no relation to data fields or variables. It is used only to visually group controls; however, it can affect the controls that are within it by means of its tab and group style settings.

The group box allows you to add a labeled box to a window. These are useful for enhancing the aesthetic quality of a window, for setting up tab stops, and for making certain groups of controls unselectable:



1. Select the group box Tool Palette button and place a group box control on top of the Salutation list box:



2. Size the group box control so that it surrounds the list box.
3. Change the group box Caption property to **Salutation** (under the HyperLabel tab) and the Name property to **SalutationGroupBox**.

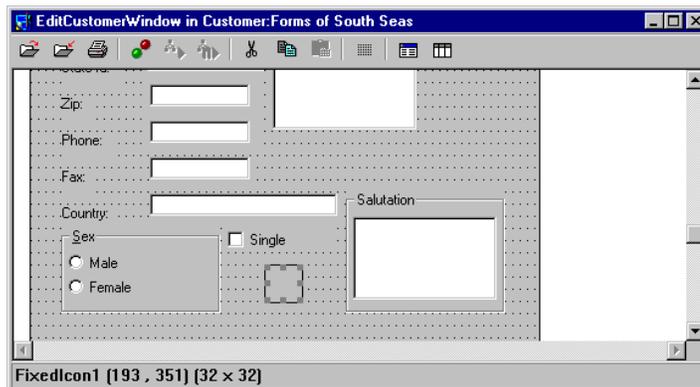
Fixed Icon Controls

The next control we will look at is the fixed icon control. This control allows you to add icons to your data windows and dialog windows. The purpose of the fixed icon control is used for aesthetic reasons only.



1. Click the Fixed Icon Tool Palette button to select it, and then move the mouse to the desired location on the window canvas area (any free spot on the window is fine).
2. Click the mouse button again.

This places a fixed icon control on the window canvas area:



3. In the Fixed Icon Properties window (under the HyperLabel tab), change the Caption property to **SSAICON**.

The Caption property holds the name of an icon that is already a part of, or associated with, your application.

Push Button Controls

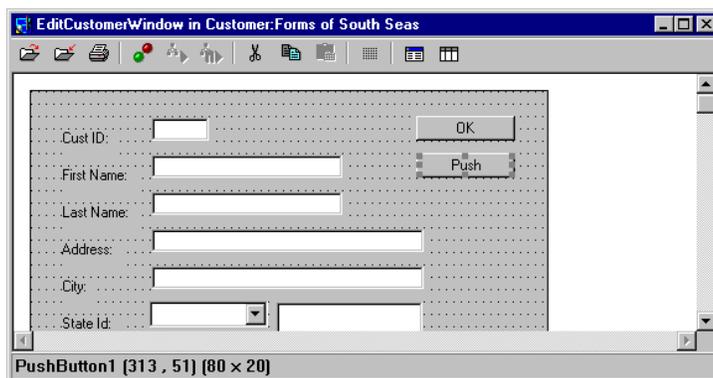
Push button controls are command controls that trigger an action without retaining any type of on/off indication. There are two types of push buttons. Standard option push buttons are the most commonly used. Default option push buttons have a slightly thicker border, and may be activated by the Enter key whenever a non-push button control has input focus. As this implies, only one push button on a dialog window should have the default option type.

Generally, a dialog window has an OK button and a Cancel button to accept or abort whatever the dialog window is trying to do. The OK button was added in the “Creating and Using Windows” chapter. Let’s add the Cancel push button to your Customer window:



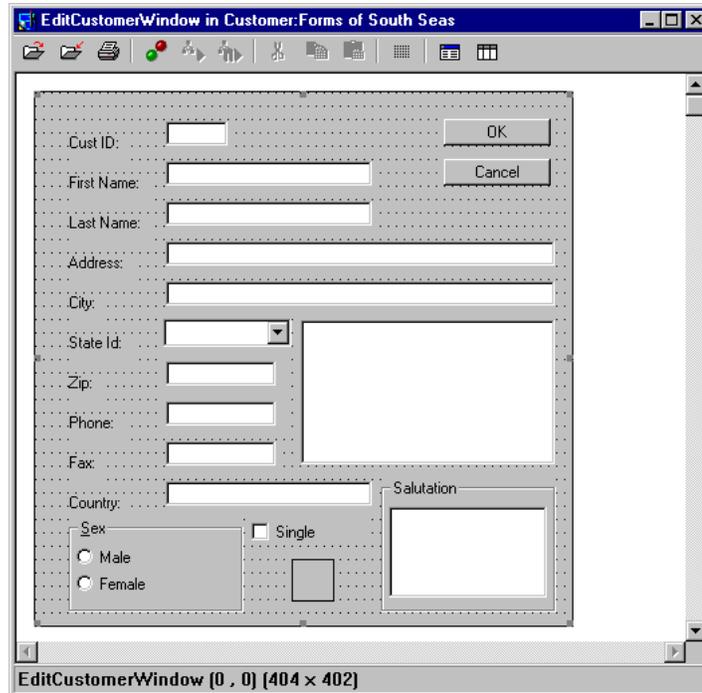
1. Select the Push Button icon on the Tool Palette and then move the mouse to the top-right corner on the window canvas area.

This places a push button onto your window canvas area, just below the OK button:



2. In the Push Button Properties (under the HyperLabel tab), change the Caption property to **Cancel**. Also, change the Name property to **CancelButton**.
3. Scroll down so you can see the bottom of the window.

4. Select the window by clicking in an area with no controls, and drag the lower border up. Adjust the list box and group box size, as well as some of the other controls so that the window looks similar to the following:



Programming Techniques

The following section deals with some of the more intricate aspects of creating controls for your windows.

Tab and Group Stops

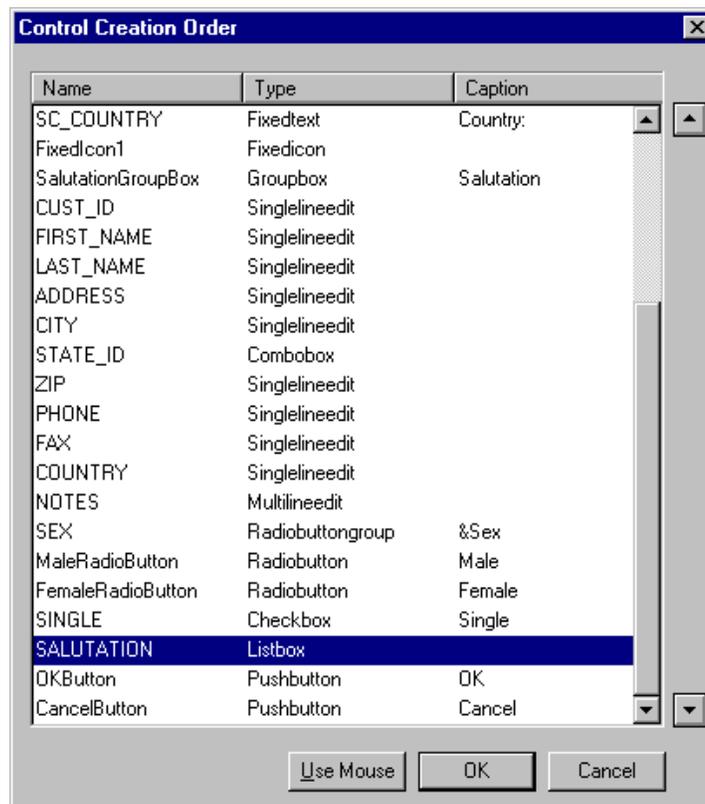
Now that you have created a data entry window, you need to control the order in which a user tabs through the controls. Of course, the Tab key moves you forward through the controls, while Shift+Tab moves back. The default tab order moves top-to-bottom, left-to-right, according to window placement. This may not always be acceptable. For example, if you had a window design consisting of two columns, you may want to move down to the bottom of the left column before moving to the top of the right.

By selecting the Control Order command from the Edit menu, you can see the order in which the controls get focus. If you scroll down to the bottom of the list, you will see the problem.

The MaleRadioButton and the FemaleRadioButton should immediately follow the Sex radio button group control. These should be promoted to reflect this. To do this, perform the following steps:

1. Click the MaleRadioButton entry and then use the up arrow button (to the right of the vertical scroll bar) to move it just below the Sex entry.
2. Click the FemaleRadioButton entry and position it just below the MaleRadioButton entry in the list.
3. Click the Single check box entry and move it up so it follows FemaleRadioButton.

Your window should now use the promote and demote buttons on the other items until it looks as follows:



4. Click the OK button to close the Control Order dialog.
5. Save the changes you've made to the EditCustomerWindow.



The order of tabbing can be forced in any sequence that you wish, based upon your specifications. But how do you decide what gets focus and what does not? The operating system provides all the logic to move input focus from one control to another, with a little help from you and the styles that can be applied to a control.

Control Order and Multiple Groups

Control order for multiple groups can also require special attention. The problem is that one group does not end until the next group begins. In short, imagine you have several radio buttons in a radio button group box. The radio button group has the group style selected.

Now let's say that you have a separate set of radio buttons. The default Windows action is to assume that this set of radio buttons belongs to the first radio button group. This is wrong, and to fix it, you would create a second radio button group for the second set of radio buttons. This would cause a new group to start, ending the first group—since every radio button group has the group style. If you do not use either a group box or a radio button group, you can set the group style for any other control so it starts a different group.

Naming Controls

Your data-entry window is complete; but before completing this chapter, let's discuss the naming convention that was used in naming the controls.

For data windows you create that need to update fields in a table, you use the field name as the control name, since a name-based association exists with the attached data server to apply changes to fields directly for you.

What about controls that are not linked to fields? They can be called anything, except the name of a field in an attached server. Long names are supported, giving you the ability to create descriptive names. The convention used named the button according to its type, followed by a string describing the type of control in question. For example:

- A push button labeled Edit is called `EditButton`
- A radio button labeled Payment is called `PaymentRadioButton`
- A single-line edit that represented a search string is called `SearchSLE`
- A list box representing the state is called `StateListBox`

Using this convention, you see that your control name tells you what the entity represents, as well as the properties you can expect it to have. Alternatively, you could use the name prefixed with initials to tell you what type of control it is. For example:

- A push button labeled Edit is called pbEdit
- A radio button labeled Payment is called rbPayment
- A single-line edit that represented a search string is called sleSearch
- A list box representing the state is called lbState

Using this convention means less typing and is just as descriptive. The only problem is that ComboBox and CheckBox both try to use cb as its prefix. The answer to this is to use cb for ComboBox and cx for the CheckBox since you could put an X in the checkbox. In this way when you look at the code a control would look similar to oDCsleSearch which is slightly easier to read.

You can now close the Window Editor. When you return to the Repository Explorer, select the Build toolbar button to compile all of the changes.

Summary

At this point, you have seen most of the window controls in use. With these tools at your disposal, you can now generate almost any kind of window for your users.¹

Inheritance and Subclassing

In this lesson, you will learn how and when to use inheritance to customize the behavior of objects in your application. You will see how CA-Visual Objects 2.7 uses inheritance to deliver a powerful development environment and, consequently, you will gain an appreciation for the benefits of subclassing and code reuse.

Overview

Inheritance is one of the fundamental principles of object-oriented programming. It allows you to specify new classes in terms of existing ones. The new class inherits all of the attributes (*instance variables*) and behavior (*methods*) of the existing class and allows you to add any distinguishing features that are needed.

Subclass, Superclass,
and Parent

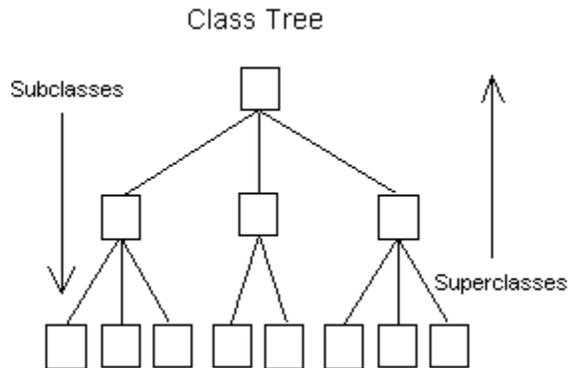
When a class inherits from another class, it is said to be a *subclass* of that class. A *superclass* is any class from which another class derives its behavior. The *parent* is the immediate superclass.

Inheritance allows for incremental development by creating new classes on the stable foundation of existing classes. In this way, a great deal of code can be reused by inheriting behavior and characteristics, and a significant level of robustness can be easily maintained.

Class Tree

CA-Visual Objects allows for each class to inherit from a single parent class. This is called single inheritance. On the other hand, each class can have as many subclasses as required.

This gives rise to a class tree, a hierarchical representation of the relationships between classes:



It is easy to see that classes defined at higher levels in the class tree are more likely to be general, and have a higher level of abstraction, while classes defined at lower levels are more specialized.

In CA-Visual Objects, a subclass is created using the INHERIT keyword in the CLASS statement. For example, to create a subclass of DataWindow called EditItemWindow, you would use the following class declaration statement:

```
CLASS EditItemWindow INHERIT DataWindow
```

EditItemWindow has all of the properties and behaviors of the DataWindow class, plus a few of its own which you will code. One way to look at EditItemWindow is that it is a *kind of* data window.

When and How to Create a Subclass

Whenever you require special behavior, first look to see if you already have a class that provides what you need. If one exists, use it. However, if there is no class that does exactly what you want, but there is one which provides the same behavior at a more basic level, then this can be used as a parent to create a subclass with the desired characteristics. In this way, subclassing is like specialization.

During the development of an application, you may find that a class is too specific to be of general use. In this case, you must create a class at a greater level of abstraction, which is more general. By doing this, you actually remove specialized behavior and attributes from the class, making it more generic. It should now be possible to create various subclasses—each with its own area of specialization—to fill the required roles. As you can see, this provides you with a very powerful and flexible development environment.

Subclassing with Generated Code

The visual editors in CA-Visual Objects, such as the Window Editor and the Menu Editor, generate source code for your application. If you examine the code created by these editors, you will notice that inheritance is used to define a new class for your purposes.

For example, the Window Editor generates a new window class definition in terms of an existing window class. This empowers the visual editors with a greater degree of flexibility and ensures that your application will be consistent and robust.

Each time you edit a binary entity, by saving the current design in one of the visual editors, CA-Visual Objects regenerates any required class definitions and supporting entities. For this reason, you should *never* modify the generated code directly, since it will be overwritten by the regenerated code from the editor.

If you wish to customize the behavior of the generated class or the supporting entities, create a subclass based on the class created by CA-Visual Objects. Any specialized behavior can be added to this new class. As you review and analyze source code in the chapters to follow, you will see this technique employed in the South Seas Adventures application.

Exercise

Customizing Generated Code

To customize generated code:

1. Open the South Seas Adventures by double-clicking its branch on the Repository Explorer tree view.
2. Open the File:Forms module by clicking its branch in the Repository Explorer tree view.

Examine the class definition created by the Window Editor by double-clicking the BaseFileDialog class entity in the Repository Explorer list view:

```
CLASS BaseFileDialog INHERIT DIALOGWINDOW

    PROTECT oCCOkButton AS PUSHBUTTON
    PROTECT oDCtheGroupBox1 AS GROUPBOX
    PROTECT oCCAdventureRadioButton AS RADIOBUTTON
    PROTECT oCCCustomerRadioButton AS RADIOBUTTON
    PROTECT oCCPaymentRadioButton AS RADIOBUTTON
    PROTECT oCCEmployeeRadioButton AS RADIOBUTTON
    PROTECT oCCItemRadioButton AS RADIOBUTTON
    PROTECT oCCInvoiceRadioButton AS RADIOBUTTON
    PROTECT oCCCancelButton AS PUSHBUTTON

    //USER CODE STARTS HERE (do NOT remove this line)
    PROTECT FileType AS USUAL
```

Notice that BaseFileDialog inherits from the DialogWindow class. This predetermines a great deal of the window's behavior and characteristics.

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

Both the Open File dialog and the New File dialog inherit from the BaseFileDialog window. To make it work properly, we have added a FileType instance variable that can be inspected once the dialog box is closed via the Access that has been written.

This class is never accessed directly but is used as the basis for the File Open window:



The functionality that is required is added to the FileOpenDialog class. This is the class that you would actually be instantiated.

Summary

Subclassing and inheritance is a very important and powerful feature of object-oriented programming. CA-Visual Objects helps you make the most of this powerful concept and allows you to quickly and easily develop robust applications that meet your business needs.

You can now move on to the next chapter, which demonstrates how to create menus and toolbars, which you can then add to your windows.

Creating Menus and Toolbars

This lesson introduces you to the basic concepts of menus and toolbars, and how they interact with your application. By the end of this lesson you will:

- Know how to create and customize menus and toolbars
- Understand how menu events invoke methods within your application
- Be able to attach menus and toolbars to your windows
- Know how to customize the behavior of menus

Overview

Menus and toolbars are essential components of a Windows application. Menus allow the user to navigate through the system and issue commands, while toolbars provide quick and easy access to frequently used menu commands within an application.

Each application can have its own menus and toolbars, which are attached to windows. CA-Visual Objects 2.7 allows you to generate and modify menus and toolbars using the Menu Editor.

Exercise

In this exercise you will create a menu and a toolbar.

Creating a New Module

Let's begin this exercise by creating a new module, for which we can then create a new menu:

1. Open the South Seas Adventures application by double-clicking its branch in the Repository Explorer tree view.
2. Create a new module by selecting the New Module toolbar button.



3. Type **Customer:Menu** in the Enter Module Name edit control.
4. Choose OK to create the new module.

The new Customer:Menu module branch appears in the South Seas Adventures tree.

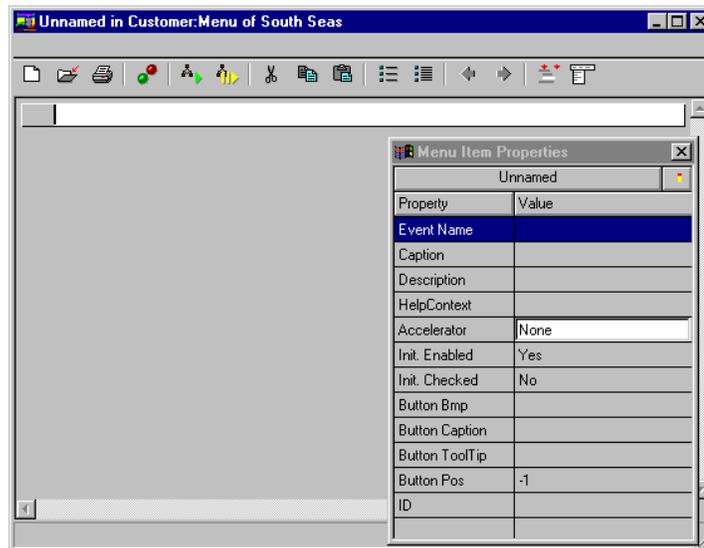
Creating the Menu

Let's create a menu:



1. Select the Customer:Menu module and select the New Entity toolbar button.
2. Choose Menu Editor from the local pop-up menu.

The Menu Editor window appears with the cursor positioned at the empty menu item:

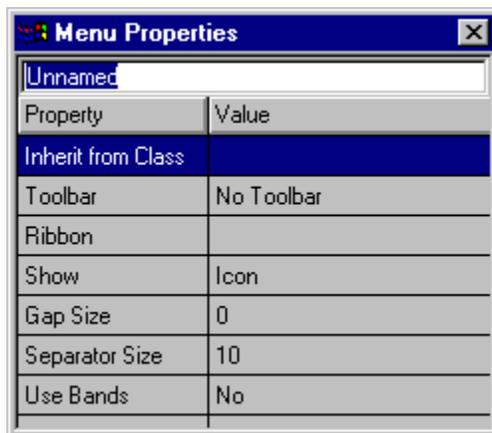


The Menu Item Properties window displays the properties associated with this item.



3. Click the Pencil icon in the Menu Item Properties window.

The properties window now displays the general properties for the menu, as indicated by the title bar:



4. Type **CustomerMenu** in the edit control and press Enter.

This assigns a name to the menu. The menu name is used to generate the menu class for your menu and used to attach your menu to a window.

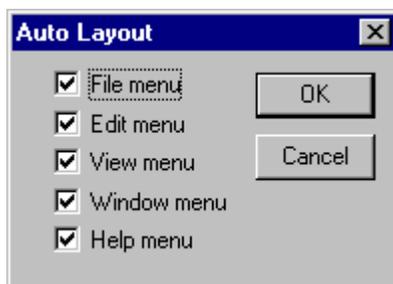
Using Auto Layout

The easiest way to define a menu is to use the Auto Layout feature of the Menu Editor. This feature allows you to add predefined menus that are commonly found in Windows applications. It also fills in many of the Menu Item Properties for each of the predefined menus.



1. Select the Auto Layout toolbar button.

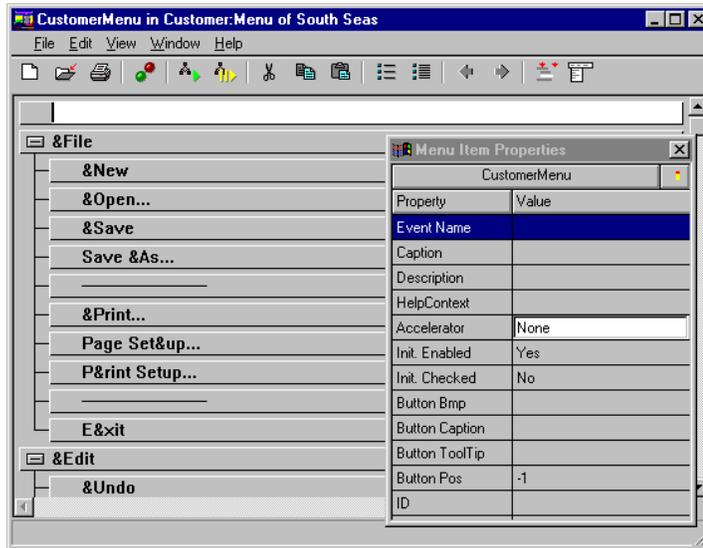
The Auto Layout dialog box appears:



Auto Layout allows you to choose from the five menus that are most often seen in Windows applications: File, Edit, View, Window, and Help. The selected menu items are appended to the menu you are currently defining.

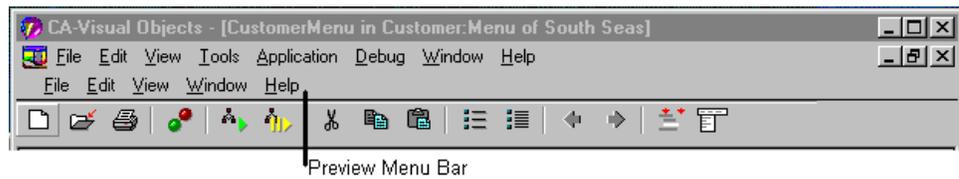
- Choose OK to accept all the predefined menus.

The menu hierarchy displays in a tree-like structure. The first level of items are those displayed on the menu bar. The indented items are the commands that are associated with the corresponding menu. For example, the File menu has the New, Open, Save, Save As, Print, Page Setup, Print Setup, and Exit commands:



Previewing Your Menu

You may have noticed that a second menu bar appears on the Menu Editor window, once you have created menu entries using Auto Layout. This can be seen in one of two ways. If you are running this window maximized, it will look like this:



If you are using normal windows, the menu bar will look like this:



The menu directly above the toolbar is the *preview menu bar*. The preview menu bar not only displays the menu you are creating, but it is a working prototype of the new menu.

Here is an example of how the preview menu bar is useful when developing your application's menu:

1. To preview the File menu from the Menu Editor, click on File in the preview menu bar. The pull-down menu associated with the File menu appears, just as it would appear in your application.
2. You can also click on some of the other menu entries of the preview menu bar to see how they appear in the application.

Collapsing/Expanding the Menu Structure

The menu structure you are currently viewing is quite long. Here are some ways to make the structure easier to view and manipulate:

1. You can control the display of a single branch of the tree by clicking its Collapse () or Expand () toggle button.
2. Simplify the display at any time by selecting the Collapse All toolbar button.



This collapses all branches of the tree. The Expand button is now present to the left of all menus with menu items.

3. To return to the full display, select the Expand All toolbar button.



The Collapse button is now present next to each item.

Adding an Item to the Hierarchy

Let's add a Report menu, which contains a command for the Customer List Report, to the predefined entries created using Auto Layout:

1. Scroll down through the Menu Editor and click the View menu.
2. Press Enter to create an empty item *after* the View menu.
3. Type **&Report**, and press Enter.

This creates the Report menu and inserts an empty item below it.

4. Type &Customer List...

Note: Do *not* press Enter. If you do, a new menu item will be inserted into the menu. You will learn how to remove this item later in the [Removing Menu Items from the Hierarchy](#) section.

Tip: Each menu and menu item in a Windows application typically features a single underlined letter that indicates how to select it from the keyboard. For example, the “R” in the Report menu and the “C” in Customer List menu entry are underlined, indicating that you can select the Customer List Report command by pressing the Alt+R, C key combination. To add this type of functionality to your menus, simply preface the letter that is to be underlined with an ampersand (&).

5. You can preview the modified menu items and their key combinations, by clicking the preview menu bar.

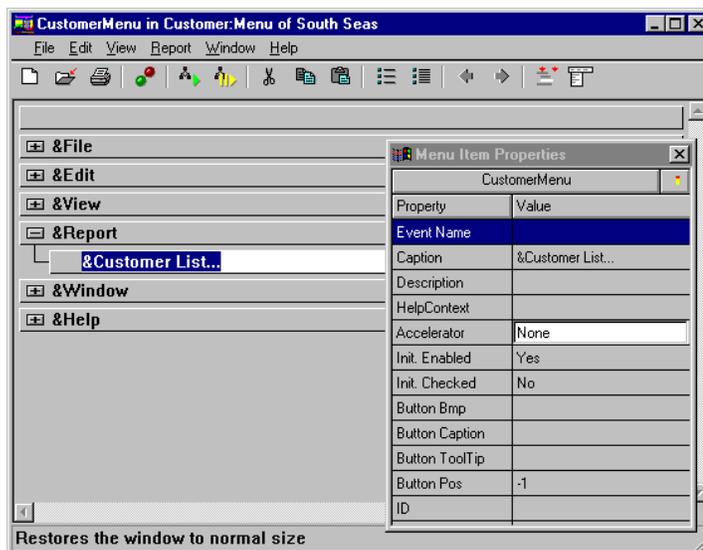
Changing the Hierarchy of a Menu Item

You now have two new entries on the same hierarchical level. The Customer List should be a child of the Report menu. You must now redefine the hierarchy between the two new items as follows:

1. Click the Customer List menu item to make it the currently selected entry.
2. Select the Demote Item toolbar button.



This makes Customer List a menu item under, or a child of, the Report menu:



3. Preview the modified hierarchy by selecting the Report menu from the preview menu bar.

Removing Menu Items from the Hierarchy

Since you do not need all the menu items created by Auto Layout, let's remove a few:

1. Choose the Save As menu command from the File menu. (If you have collapsed your menus, expand them to select the Save As menu item.)
2. Delete the item by selecting the Delete Item command from the Edit menu (or by pressing Ctrl+Y).
3. Using the two previous steps, delete the following menu items:
 - The empty menu item above the File menu that was initially created as the first item in the Menu Editor
 - From the File menu: Save, Print, and Page Setup
 - From the Edit menu: the first separator line, Insert, Delete, Go To, and Find Next

From the View menu: the separator line, All Records, and Select Records

Specifying Menu Actions to Perform

For a menu item to perform an action in your application, you must specify an Event Name. If you review the menu items you created with the Auto Layout feature, you notice that each contains an Event Name in its Menu Item Properties window.

The Event Name property specifies the name of a method to call, which the menu attempts to locate in several ways. First, the menu attempts to call this method from within its owner window. If its owner window has the method defined or is subclassed from a window that has it defined, the method is called. If no method exists in the owner window, it attempts to call it from this window's owner. It continues to call all owners until it reaches the application level.

If none of the owners within the window ownership hierarchy have this method defined, the menu searches for a Window class of the same name. If one is located, the menu opens the window.

If a Window class is not found, it then searches for a ReportQueue class of the same name. If one is found, it is executed.

If a ReportQueue class is also not found, the MenuCommand() method (a Windows callback) provides default behavior so that your applications can be prototyped and compiled successfully during development.

The fact that nothing happens (which also means there is no runtime error) makes CA-Visual Objects a great prototyping tool. You can define your menus first and make them functional incrementally.

SSAWindow Event Name

Now, let's use a method of the South Seas Adventures SSAWindow as the Event Name for the Customer List command:

1. Select the Customer List menu item, under the Report menu, in the Menu Editor.
2. Click the Event Name property of the Menu Item Properties window.
3. Type **CustomerReport** in the Value edit control and press Enter.

This particular menu is to be attached, later in this lesson, to the EditCustomerWindow. The EditCustomerWindow does not have a CustomerReport() method. However, the EditCustomerWindow's owner, SSAWindow, does. It is this method that is invoked when the Customer List menu item is selected.

Providing Menu Shortcuts

You may further simplify your application by providing an *accelerator* (or shortcut key). An accelerator is usually attached to frequently used menu commands. It allows the user to press a specific key combination to directly access a command—bypassing the menu altogether.

To provide this functionality, you need to specify a key, or a key combination, in a menu item's Menu Item Properties window. CA-Visual Objects automatically appends the name of the accelerator key to the menu command caption.

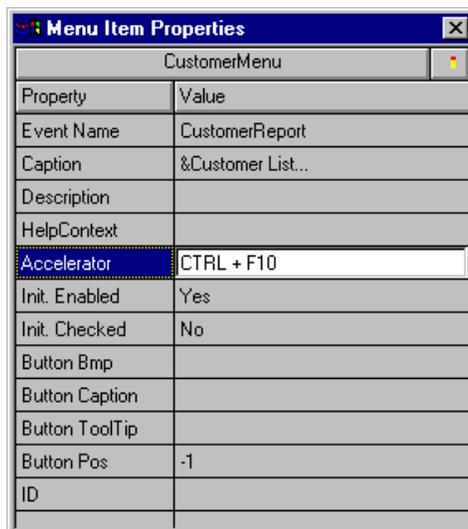
Let's define Ctrl+F10 as the accelerator key for the Customer List Report command:

1. Select the Customer List menu item under the Report menu.
2. In the Menu Item Properties window, click the Accelerator property (not the value) to put focus on this property.

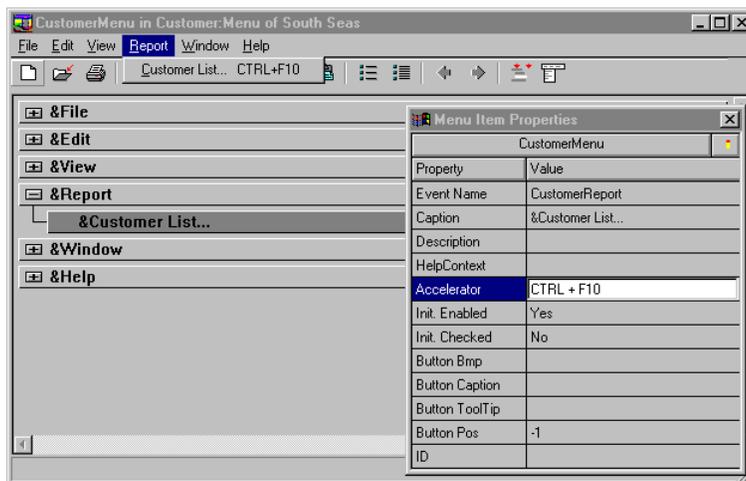
Note: To activate the edit control in the Value column, click the Accelerator property column first. Also, for this property to take effect, do *not* press Enter after entering the accelerator key. Instead, move the cursor and click a *different* property outside of the Value column.

Hold down the Ctrl key and press the F10 key.

The Properties window should now look like this:



To view the accelerator key for the Customer List menu item, select the Report menu from the preview menu bar, as seen below:

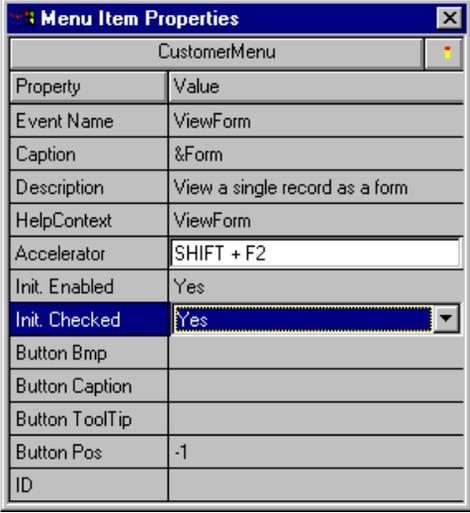


Checking a Menu Item

Your menu can be defined to provide a visual cue for a current selection—by displaying a check mark next to your menu command. For example, the menu you are currently defining is to be attached to a window that initially displays data in form view. You can place a check mark next to the Form menu item to indicate the current view state for your user by following these steps:

1. Select the Form menu item, under the View menu, in the Menu Editor.

2. In the Menu Item Properties window, click the Init. Checked property. Select Yes from the list box in the Value cell, as seen below:



CustomerMenu	
Property	Value
Event Name	ViewForm
Caption	&Form
Description	View a single record as a form
HelpContext	ViewForm
Accelerator	SHIFT + F2
Init. Enabled	Yes
Init. Checked	Yes
Button Bmp	
Button Caption	
Button ToolTip	
Button Pos	-1
ID	

Later in this lesson, when a different view is selected, the check mark from the Form View command is removed.

Creating a Toolbar

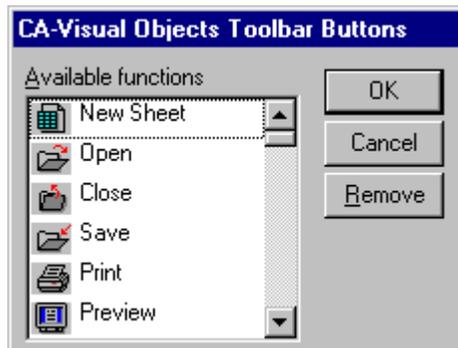
As stated earlier, toolbars are attached to windows, yet you create them using the Menu Editor. The reason for this is that CA-Visual Objects associates the buttons of a toolbar with menu commands used in the application. Typically, only a few menu commands are depicted on the toolbar—those that are most frequently used.

Create Toolbar Buttons

When you originally create a menu, an empty toolbar is associated with this menu. To create toolbar buttons, define the Button property for the menu items to be accessed from the toolbar:

1. Select the Customer List menu item, under the Report menu, in the Menu Editor.
2. Click the Button property in the Menu Item Properties window.

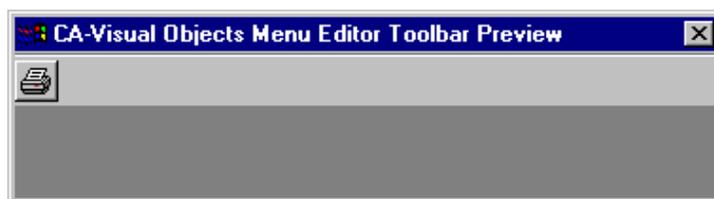
The Toolbar Buttons dialog box appears:



3. Select the Print icon from the list and choose OK.

The Print icon is now displayed in the Value column along with the caption that appears on the status bar when the user moves the cursor over the icon. In the Menu Item Properties window, the Button Pos value is now 1. This indicates that this is the first button on the toolbar.

4. Select the Preview Toolbar command from the Menu Editor's File menu to preview the toolbar you are creating:



5. Double-click the Menu Editor Toolbar Preview window's system menu to return to the Menu Editor.
6. Select the Next menu item under the Edit menu.
7. Click the Button property in the Menu Item Properties window.
8. Scroll down through the list of available options (near the bottom) to select the Next Record icon from the Toolbar Buttons dialog box and then choose OK.

The Button Pos value is 2, indicating that the Next Record icon is the second button on the toolbar.

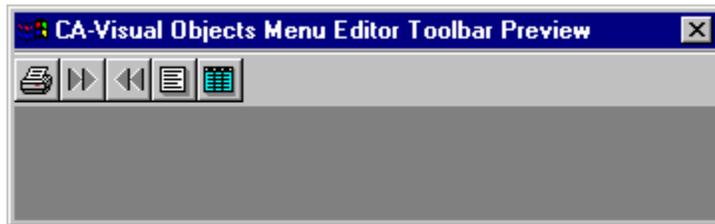
9. Select the Previous menu item under the Edit menu.
10. Click the Button property in the Menu Item Properties window.

11. Scroll down through the list of options (once again, near the bottom) to select the Previous Record icon from the Toolbar Buttons dialog box, then choose OK.

The Button Pos value is 3, indicating that the Previous Record icon is in the third button on the toolbar.



12. Select the Form menu item under the View menu.
13. Using steps 10 and 11, assign the View Form icon (at the bottom of the list) to the Button.
14. Select the Table menu item, under the View menu.
15. Assign the Table icon (near the top of the list) to the Button BMP. Do not use the View Table bitmap from the bottom of the list.
16. Select the Preview Toolbar command from the Menu Editor's File menu to preview the completed toolbar:



17. Double-click the Menu Editor Toolbar Preview window's system menu to return to the Menu Editor.

Changing Toolbar Button Positions

You may reorder the position of toolbar buttons, once created, by changing the Button Pos value in the Menu Item Properties window.

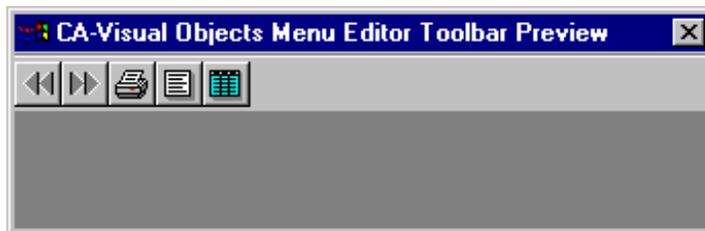
1. Select the Customer List menu item, under the Report menu.
2. Click the Button Pos property in the Menu Item Properties window and change the value to **3**.

This moves the Printer button assigned to this command to the third position.

3. Select the Previous menu item under the Edit menu.
4. Click the Button Pos property in the Menu Item Properties window and change the value to **1**.

This moves the Previous Record button assigned to this command to the first position.

5. Select the Preview Toolbar command from the Menu Editor's File menu to preview the modified toolbar:

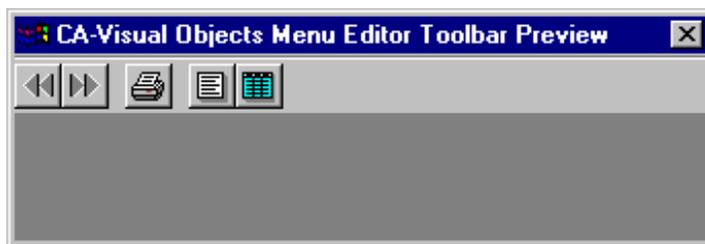


6. Double-click the Menu Editor Toolbar Preview window's system menu to return to the Menu Editor.

Spacing Between Toolbar Buttons

You may also modify the gaps, between buttons or a group of buttons, for your toolbar. This is achieved by skipping a number when assigning button positions as detailed below:

1. Select the Customer List menu item under the Report menu.
2. Click the Button Pos property in the Menu Item Properties window and change the value to **4**.
This leaves a gap before the Printer button.
3. Select the Form menu item under the View menu.
4. Change the Button Pos to **6**, to leave a gap before the ViewForm button.
5. Select the Table menu item under the View menu and change the Button Pos to **7**.
6. Select the Preview Toolbar command from the Menu Editor's File menu to preview the modified toolbar:



Notice the new icon positions and the gaps before and after the third icon.

7. Double-click the Menu Editor Toolbar Preview window's system menu to return to the Menu Editor.

Other Modifications to the Toolbar

In addition to changing button positions and modifying the spacing between buttons, toolbars have several other properties that can also be modified.

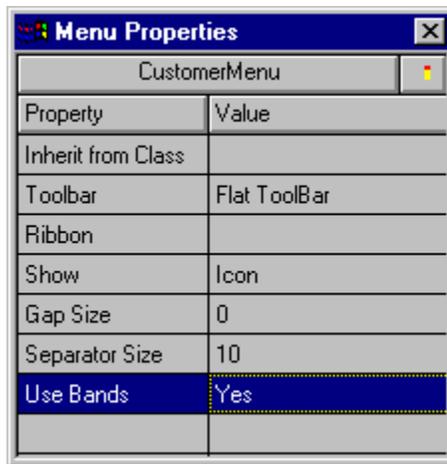
Let's keep the toolbar style as Flat Toolbar, which is the Internet Explorer style, and introduce the **Bands** as the separator (consistent with the Internet Explorer style of toolbar you have created):



1. Click the Menu Properties toolbar button in the Menu Item Properties window (or select the Menu Properties command from the Edit menu).

The Menu Properties window displays.

2. Click the Use Bands property and change it to **Yes**:



Note: The Preview Toolbar window will not show any difference but it will become visible in the application.

For a description of other Menu Properties, see Using the Menu Editor topic, in online help.

Saving the Menu

When you have completed your menu definition, you must make it available to the Window Editor. To do this, follow these steps:



1. Click the Save toolbar button.



2. Select the Build toolbar button.

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

Attaching a Menu to a Data Window

Because the `EditCustomerWindow` has a different menu attached to it, you will have to make a change to its `ViewToggle()` method, which is in the `Tutorial:Windows` module. This module was imported in the "Creating and Using Windows" chapter as `TUTWIND.MEF`.

The `View Toggle()` method is one of the few developer-coded menu event methods. It modifies the state of menu items by using the `CheckItem()` and `UncheckItem()` methods. These methods require you to specify the name of a constant, such as `IDM_SSACHILDMENU_VIEW_TABLE_ID`, to identify the menu items. Since the `EditCustomerWindow` now utilizes the new `CustomerMenu`, these constants must be changed. To do this:

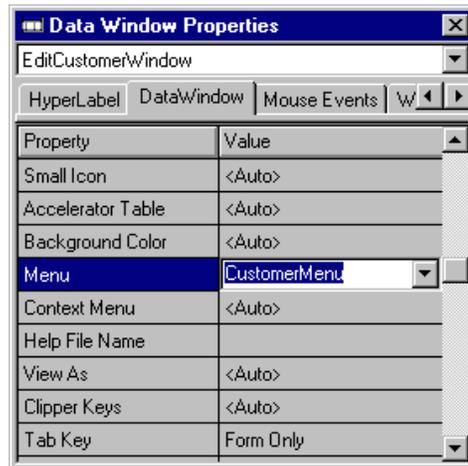
1. Open the `Tutorial:Windows` module by clicking its branch on the Repository Explorer.
2. Find the `EditCustomerWindow:ViewToggle()` method and double-click it to open the Source Code Editor.
3. Select the Edit Replace command.
4. Type `SSACHILDMENU` in the Find What edit control and type `CUSTOMERMENU` in the Replace With edit control.
5. Choose Replace All.

You will see that the new substring appears in several different lines.



6. Save your changes by selecting the Save toolbar button.
7. Rebuild the application by selecting the Build toolbar button.
8. Close the Source Code Editor by double-clicking its system menu.
9. Change to the `Customer:Forms` module and open the `EditCustomerWindow` in the Window Editor.
10. Change to the `DataWindow` tab in the properties window and click on the Menu property to drop a list of the available menus that are available.

11. Select CustomerMenu from the list:



12. Save your changes by selecting the Save toolbar button.

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

Putting It All Together

You are now going to run and test the application with the changes you have made:



1. Rebuild the application by selecting the Build toolbar button.



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

3. Choose OK at the opening dialog box.

The Login dialog box appears.

4. Type **user** in the Name edit control and **trainee** in the Password edit control. Choose OK.

5. Select the Open command from the File menu.

6. Click the Customer radio button and choose OK. The Customer Browser displays.

7. Select a customer by clicking one of the cells in the Customer Browser and choose Edit. (This functionality was described in [Chapter 5: Creating and Using Windows](#))

The Edit Customer window displays with the menu and toolbar you created for it.

View your modified menu, by selecting the Report and/or the Edit menu. You are now able to see the results of setting the Use Bands property:



8. Close the application by double-clicking its system menu.

Designing a Menu

When designing your application, you face a decision about the number of menu definitions that are required. One possible approach is to create a menu for each window of your application.

However, in many applications you find that most actions required in windows are common to many windows. Therefore, defining a separate menu for each window is redundant. Another approach is to find common functionality between your menus and to manage the exceptions.

The SSACChildMenu

Before you created the CustomerMenu menu, you may have noticed that the South Seas application uses only one menu for all the child windows.

Since most windows in the South Seas Adventures application fall into one of two categories—the Edit or New windows—only one menu (SSACChildMenu) has been created to be used by most data windows. The actions defined in this menu are generic enough to be used and unchanged by each Edit window. As for the New windows, some of the commands such as “movement” commands have been disabled.

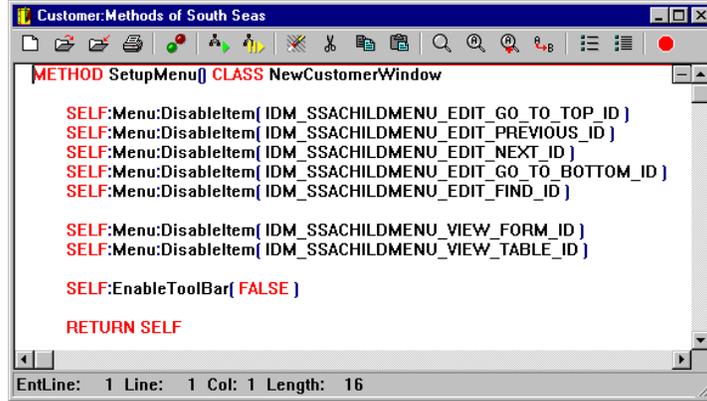
Customizing a Menu

In certain instances, you may want to further customize your menu. For example, the NewCustomer window does not require that movement in the data server be allowed while a customer is being added. It also does not allow switching between Form and Table view. In addition, since most toolbar buttons are movements, the toolbar is disabled as well.

Disabling Menu Items

The Setup() and SetupMenu() methods for the NewCustomer window have been created to make the desired menu modifications. Whenever a NewCustomer window is opened, the SetupMenu() method is called by its Init() method.

You can find the source code for the SetupMenu() method in the Customer:Methods module of the South Seas Adventures application, as shown below:



```
METHOD SetupMenu() CLASS NewCustomerWindow

SELF:Menu:DisableItem( IDM_SSACHILDMENU_EDIT_GO_TO_TOP_ID )
SELF:Menu:DisableItem( IDM_SSACHILDMENU_EDIT_PREVIOUS_ID )
SELF:Menu:DisableItem( IDM_SSACHILDMENU_EDIT_NEXT_ID )
SELF:Menu:DisableItem( IDM_SSACHILDMENU_EDIT_GO_TO_BOTTOM_ID )
SELF:Menu:DisableItem( IDM_SSACHILDMENU_EDIT_FIND_ID )

SELF:Menu:DisableItem( IDM_SSACHILDMENU_VIEW_FORM_ID )
SELF:Menu:DisableItem( IDM_SSACHILDMENU_VIEW_TABLE_ID )

SELF:EnableToolBar( FALSE )

RETURN SELF

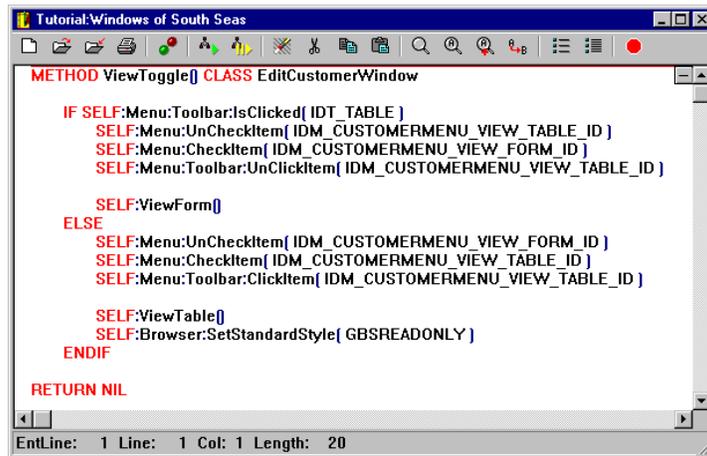
EntLine: 1 Line: 1 Col: 1 Length: 16
```

Note: When a NewCustomer window displays, the disabled items appear dimmed on its menu.

Editing Toolbar Buttons

You can modify the standard behavior of toolbar buttons using methods of the menu class. The Toolbar Buttons dialog box displays with the View Form icon highlighted.

When the Table view is selected from the menu command or with the button, the button displays with a depressed shape. This is accomplished with the ViewToggle() method of the EditCustomerWindow class. This method is contained in the Tutorial:Windows module of the application:



```
METHOD ViewToggle() CLASS EditCustomerWindow

IF SELF:Menu:ToolBar:IsClicked( IDT_TABLE )
SELF:Menu:UnCheckItem( IDM_CUSTOMERMENU_VIEW_TABLE_ID )
SELF:Menu:CheckItem( IDM_CUSTOMERMENU_VIEW_FORM_ID )
SELF:Menu:ToolBar:UnClickItem( IDM_CUSTOMERMENU_VIEW_TABLE_ID )

SELF:ViewForm()
ELSE
SELF:Menu:UnCheckItem( IDM_CUSTOMERMENU_VIEW_FORM_ID )
SELF:Menu:CheckItem( IDM_CUSTOMERMENU_VIEW_TABLE_ID )
SELF:Menu:ToolBar:ClickItem( IDM_CUSTOMERMENU_VIEW_TABLE_ID )

SELF:ViewTable()
SELF:Browser:SetStandardStyle( GBSREADONLY )
ENDIF

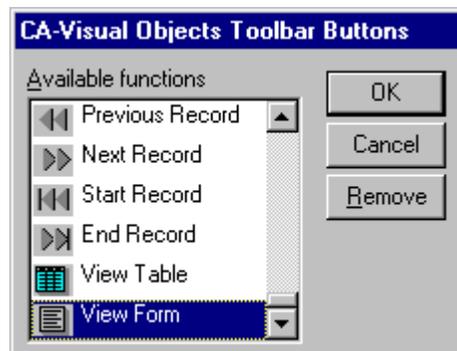
RETURN NIL

EntLine: 1 Line: 1 Col: 1 Length: 20
```

To use this method in your menu, you change the ViewTable event name to ViewToggle and remove the ViewForm button from the toolbar. To do this, you need to go back to the menu you created earlier in this lesson:

1. Click the Customer:Menu module branch in the Repository Explorer tree view.
2. Invoke the Menu Editor by double-clicking the CustomerMenu menu entity in the Repository Explorer list view.
3. Select the Table menu item, under the View menu.
4. In the Menu Item Properties window, select the Event Name property and type **ViewToggle**.
5. Select the Form menu item under the View menu.
6. In the Menu Item Properties window, select the Event Name property and type **ViewToggle**.
7. In the Menu Item Properties window, click the Button property.

The Toolbar Buttons dialog box displays with the Table icon highlighted:



8. To remove the ViewForm button from the toolbar, choose Remove.
9. Save your changes by selecting the Save toolbar button.
10. Build the application by selecting the Build toolbar button.



To view the results of your changes, follow the steps described earlier, in Putting it All Together, to execute the application and access the EditCustomer window.

Summary

In this chapter you have learned how to use the Menu Editor to create menus and toolbars, change a menu and its menu item properties, prototype menus and toolbars, and define accelerator keys. You have also learned how to modify the behavior of menus by programming new behavior.

In the following lesson you will learn how to access data from controls, data servers, and data windows and gain an understanding of how data is processed in CA-Visual Objects 2.7.

Accessing and Updating Data

This lesson demonstrates how to access data from controls, data servers, and data windows.

Overview

So far, you have seen that data windows with attached data servers provide mechanisms for updating the database and controls automatically. In this case, the programmer has nothing to code.

In many instances (for example, when creating event handlers), you will want to access data from your source code. There are many ways to access data—through methods, functions, data servers, data windows, and controls.

In this lesson, you will see how to retrieve and set data in databases and controls. You will also see that there are different ways to get at the same data.

The information in this lesson is essential to understanding of how data is processed in CA-Visual Objects 2.7.

Narrative

Xbase Compatibility

A subset of the CA-Visual Objects language is derived from Xbase. All of the usual commands work to access and assign your data. For example:

```
// Access the Amount field  
nAmount := Invoice->Amount  
  
// Assign 5 to the Amount field  
Invoice ->Amount := 5
```

Access and Assign Methods

Access and assign methods are special methods of classes. They are executed automatically each time you access data from or assign data to a named instance variable. The South Seas application contains about 500 access and assign methods. They are generated automatically by the IDE editors (Window, DB Server, and SQL).

You can define several types of instance variables in a CLASS declaration, including EXPORT, INSTANCE, HIDDEN, and PROTECT. All of these, except EXPORT, are not directly accessible externally (that is, outside of the class). The EXPORT instance variable is accessible outside of the class definition.

To access or assign to a non-exported instance variable from outside of the class, you must use a method. Indeed, this is the purpose of not exporting the variable.

The syntax for referencing a method is obviously different from that of referencing a variable. This violates encapsulation since users of the class must be aware of how a property of the class is implemented in order to know whether to use function style or variable style reference.

Using Protected Variables

In the following example, the Size instance variable is a PROTECT type variable. It therefore cannot be accessed or assigned from outside of the class:

```
CLASS Square
  PROTECT Size

METHOD Init(oSize) CLASS Square
  // Assign an instance variable (of any type)
  Size := Dimension{oSize}

RETURN SELF
```

Using a Method

You can also create a method to set the size, as in:

```
METHOD SetSize(oSize) CLASS Square
  IF oSize != NIL
    Size := oSize
  RETURN Size
ELSE
  RETURN Size
ENDIF
```

To access or assign the size of the square from outside of the class, you would then code:

```
oSquare := Square{100}

// Print the size of the square
? oSquare:SetSize()

// Change the size of the square
oSquare:SetSize(200)
```

Using Access and Assign Methods

A better way to do this would be through the use of access and assign methods. Access and assign methods allow you to reference your non-exported variable while maintaining the syntax established for instance variables. The Size access and assign methods are shown below:

```
ACCESS Size CLASS Square
RETURN Size

ASSIGN Size(uValue) CLASS Square
    Size := uValue

RETURN Size
```

From outside of the class, you can now code:

```
oSquare := Square{100}
// Print the size of the square
? oSquare:Size

// Change the size of the square
oSquare:Size := 200
```

EXPORT instance variables are accessible outside of the class definition. Why then, should you ever bother with access and assign methods?

Access and assign methods provide a layer above the implementation of the variables in question. Thus, if the implementation changes at the lower level, the program interface can remain intact. When using the variables directly, you take a risk in assuming the implementation will never change. If a change does occur, you may have to change each and every instance of the variable within your program.

Generated Data Server Classes

When you save a data server, the DB Server Editor (or the SQL Editor) creates access and assign methods for each field in the table, as in:

```
ACCESS Amount CLASS Invoice
RETURN SELF:FieldGet(#Amount)

ASSIGN Amount(uValue) CLASS Invoice
RETURN SELF:FieldPut(#Amount, uValue)
```

You can, therefore, access and assign data from the table by coding:

```
// Access Amount field
nAmount := oServer:Amount

// Assign 5 to the Amount field
oServer:Amount := 5.00
```

You can also access and assign fields of the data server by using the data server methods `FieldGet()` and `FieldPut()` directly, as the access and assign methods did above:

```
// Access Amount field
nAmount := oServer:FieldGet(#Amount)

// Assign 5 to the Amount field
oServer:FieldPut(#Amount,5)
```

Both coding styles yield identical results; however, using the access and assign methods make your code more readable.

Base DBServer and SQL Classes

When using the `SQLSelect` or `DBServer` classes directly, no access and assign methods exist for the fields in the database. You might, therefore, expect an error to be generated with the following code:

```
// Create a data server
oConn := SQLConnection{"Accounting","UserID",;
"Password"}
oServer := SQLSelect{"SELECT * FROM Invoice",oConn
```

Or:

```
// Create a data server
oServer := DBServer{"Invoice"}
```

Then,

```
// Access Amount field
nAmount := oServer:Amount

// Assign 5 to the Amount field
oServer:Amount := 5.00
```

No error results because of an error interception mechanism provided by CA-Visual Objects. This mechanism is designed specifically to handle references to nonexistent instance variables. When you make a reference to an instance variable (in any class) that does not exist, a method call, either `NoIVarGet()` or `NoIVarPut()`, is made to the class in question.

The `NoIVarGet()` method is invoked when you attempt to retrieve a value from a nonexistent variable. The method is passed a parameter indicating the name of the variable to be retrieved.

The following code:

```
? oAnyClass:NonExistantVariable
```

Produces:

```
? oAnyClass:NoIVarGet(#NonExistantVariable)
```

The base data server classes use this feature to allow you to access the fields of the table to which they are associated. The `NoIVarGet()` method tests to see if a field in the table corresponds to the passed symbol name, and if so it performs a `SELF:FieldGet(<symName>)`. Similarly, the `NoIVarPut()` method tests to see if a field in the table corresponds to the passed symbol name, and if so it performs a `SELF:FieldPut(<symName>, <Value>)`.

Data Forms

This section applies to both `DataWindow` objects and subform controls. The subform control object is actually a data window and, therefore, can be used in the same way as other data windows.

When you save a data form in the Window Editor, access and assign methods are created for each **data** control on the window that **could be** linked to a data server field:

```
ACCESS Amount() CLASS PaymentSubForm
RETURN SELF:FieldGet(#Amount)

ASSIGN Amount(uValue) CLASS PaymentSubForm
SELF:FieldPut(#Amount,uValue)

RETURN Amount := uValue
```

A data control is one that can be linked to a field in an attached data server, such as an edit control or check box. The data window uses name-based linkages when deciding whether to use data from a control or from the data server. That is, it attempts to establish a linkage between the name of the control and the name of a field in the data server. Consider the following:

```
// Access Amount field
nAmount := oDw:Amount

// Assign 5 to the Amount field
oDw:Amount := 5.00
```

If the `Amount` field exists in the data server, the `FieldGet()` and `FieldPut()` methods retrieve and set the data server field, respectively. Otherwise, they retrieve and set the window's control value.

You can also use the `FieldGet()` and `FieldPut()` methods directly, as follows:

```
// Access Amount field
nAmount := oDw:FieldGet(#Amount)

// Assign 5 to the Amount field
oDw:FieldPut(#Amount, 5)
```

Both coding styles yield identical results; however, using the access and assign methods make your code more readable.

Data Servers Attached to Data Forms

Data servers attached to data windows are accessible by using the `DataWindow:Server` instance variable. They can be used as described in the Data Server Classes section above. For example:

```
// Access Amount field
nAmount := oDW:Server:Amount

// Assign 5 to the Amount field
oDW:Server:Amount := 5
```

Or:

```
// Access Amount field
nAmount := oDW:Server:FieldGet(#Amount)

// Assign 5 to the Amount field
oDW:Server:FieldPut(#Amount, 5)
```

Controls

Controls can be placed on data windows, dialog windows and data dialog windows. Controls hold data that the user inputs in a buffer. In many cases, you want to manipulate these values directly.

As you have seen, data windows automate the process of transferring data between an attached server and its controls, provided the names of the controls have corresponding fields in the data server.

If a control has no corresponding field in a data server, the control simply acts as a buffer and holds the value that the user inputs. The data window essentially leaves it alone. In this case, you have to write code to access the control's value if you want to use it.

Consider appending records to a database. In database applications, it is standard practice to buffer the user input prior to appending records. If the user chooses to save (for example, by clicking the OK button), then and only then, do you append the record. You would then update the new record manually.

For dialog windows, you may want to assign initial values to controls. For example, in a Print Report dialog, you may want the default destination to be the printer. In this case, the Printer radio button should be selected.

Whatever the reason, you will undoubtedly be required to directly access controls somewhere along the development trail. Earlier, it was stated that the Window Editor creates access and assign methods for each of its controls that **could** be linked to the connected data server (such as edit controls and check boxes). That includes those controls that are not linked to a data server field as well.

For example, to examine the access and assign methods for the `nAmount` data control of the `NewPaymentWindow` window, open the `Payment:Forms` module. The only difference from the code shown earlier for the linked control (`Amount`) is that the buffered control name (`mAmount`) is used.

If a control on a data window does not have a corresponding field in the attached data server or if the data window does not have a data server attached, the data window can still help you access and assign the values in the control.

The `CA-Visual Objects 2.7` control classes allow you to manipulate window controls. When you save a window, the Window Editor generates a class for your new window. The class entity includes instance variables for each data control on the window. The instance variables for data controls are prefixed with `oDC` (for Data Controls), such as:

```
PROTECT oDCFirst_Name
```

After instantiation, the instance variables are set to an object of the class that represents the control. For example, `oDCFirst_Name` is a `SingleLineEdit` object.

Each control class has a `Value` and `TextValue` access and assign method.

Value Access and Assign Methods

The `Value` instance variable represents the value held in the control, in whatever data type the control holds. The data type is determined by the field specification attached to a control. The field specification can be specified or assumed if it is attached to a data server field:

```
// Prints "N" for numeric
? ValType(oDCAmountControl:Value)

// Access the control
nAmount := oDCAmountControl:Value

// Assigns 5 to the control
oDCAmountControl:Value := 5
```

TextValue Access and Assign Methods

For edit controls and scroll bars, the `TextValue` instance variable represents the value held in the control as a string. The value is formatted according to the field specification attached to the control. The field specification can be specified or assumed if it is attached to a data server field:

```
// Prints "C" for character
? ValType(oDCAmountControl:TextValue)

// Access the control
cAmount := oDCAmountControl:TextValue

// Assigns "5" to the control
oDCAmountControl:TextValue := "5"
```

For other data controls—like check boxes, combo boxes, list boxes, and radio button groups—`TextValue` may contain different information than `Value`.

Summary

In this lesson, you learned about the various ways of accessing and updating data from within your program. You have also learned how to manipulate data using methods, functions, data servers, data windows, and controls.

Proceed to the next lesson to learn about using window event handlers—giving your application the ability to respond to an event initiated by the user, such as a push button click or an edit control modification.

Customizing Window Event Handlers

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 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 2.7
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 to (invokes a method of) 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 multi-line 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.

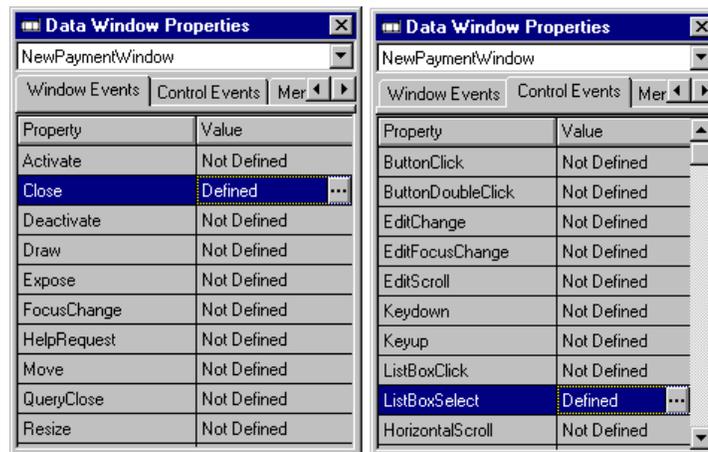
Exercise

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 its branch in the Repository Explorer tree view.
2. Open the Payment:Forms module by clicking its branch in the Repository Explorer tree view.
3. Open the NewPaymentWindow window by double-clicking on this entity in the Repository Explorer list view.

The Window Editor displays.

4. In the Data Window Properties window, you can see that the Close event has been defined on the Window Events tab and the ListBoxSelect event has been defined on the Control Events tab:



If you want to customize the behavior of a window, just redefine the event methods that you wish to change—it's that simple. Several events that you may want to customize are described below:

- 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.

Using the EditChange() Method

The method we are about to write will print the amount, in word format, across the payment receipt as the amount is entered in the amount single-line edit control.



1. Click the EditChange property in the Data Window Properties window(Control Events tab), then click the ellipsis button.

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

```
METHOD EditChange( oControlEvent ) CLASS;
    NewPaymentWindow
    LOCAL oControl AS Control
    LOCAL uValue AS USUAL
    oControl := IIf( oControlEvent == NULL_OBJECT, ;
        NULL_OBJECT, oControlEvent:Control)
    SUPER:EditChange( oControlEvent )
    //Put your changes here
    RETURN NIL
```

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

```
METHOD EditChange( oControlEvent ) CLASS;
    NewPaymentWindow
    LOCAL oControl AS Control
    LOCAL uValue AS USUAL
    oControl := IIf( oControlEvent == NULL_OBJECT, ;
        NULL_OBJECT, oControlEvent:Control)
    SUPER:EditChange( oControlEvent )
    IF oControl: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, oControlEvent, 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 oControl:= oControlEvent:Control
```

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

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

```
oControl: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 oControl: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(oControlEvent)
```

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 its system menu.

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



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



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

Viewing Your Results

Verify the results using the following steps:

1. Select the New command from the File menu.

The New Record dialog box appears:

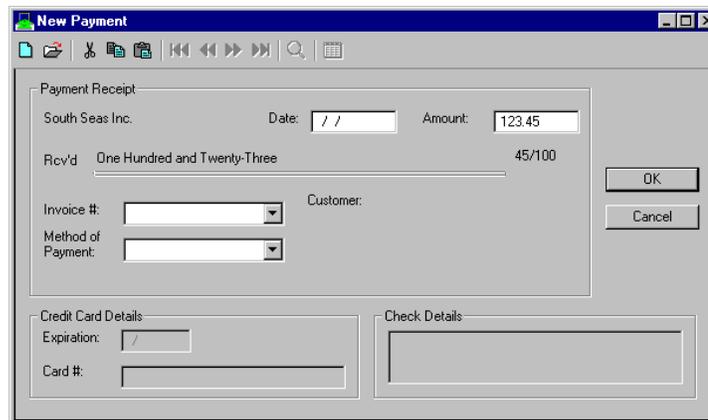


2. Select the Payment radio button and then choose OK.

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 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, check boxes and push buttons.

Using the Notify() Event Handler

The NotifyEvent event only occurs for data-aware windows and is created by an attached server. This event is crucial for a data window, since the data aware windows use 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-aware 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-aware window that it has repositioned the record pointer. The data-aware window then updates its controls from the server.

Your program can make use of this mechanism.

Creating the Method

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 clicking its branch in the Repository Explorer tree view.
2. Find the EditItemWindow: Notify() method and double-click its entity to display the following code:

```
METHOD Notify( kNotifyName, uDescription ) CLASS ;
    EditItemWindow

    LOCAL xRetNotify := SUPER:Notify ( ;
        kNotifyName, uDescription )
    STATIC LOCAL lRecordUpdated := FALSE AS LOGIC

    // 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
        lRecordUpdated := TRUE
    ENDIF

    IF kNotifyName == NOTIFYRECORDCHANGE .AND. ;
        lRecordUpdated == TRUE
        SELF:Owner:BroadcastMessage( SELF, #ITEM )
        lRecordUpdated := FALSE
    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 to identify any event that involves a field change:

```
IF kNotifyName >= NOTIFYFIELDCHANGE
```

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

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)
```



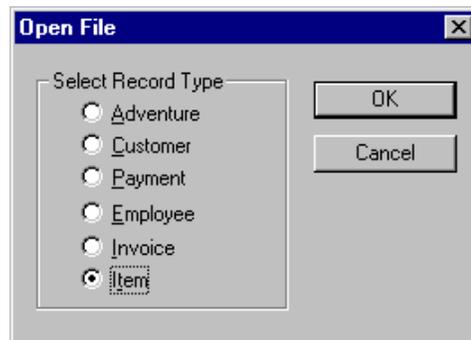
3. Run the application by selecting the Execute toolbar button.

Viewing Your Results

Verify the results, using the following steps:

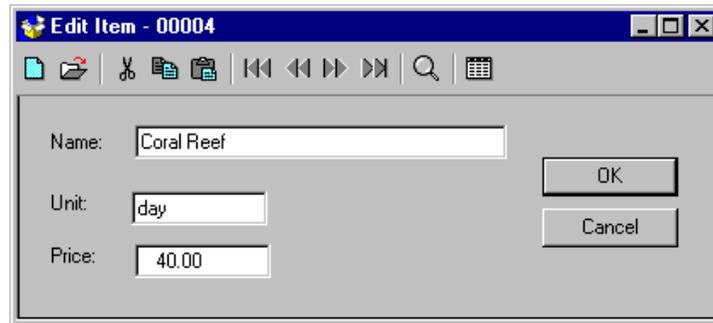
1. Select the Open command from the File menu.

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 the Edit push button.

The Edit Item window appears:



4. To verify that the Notify() method is working, click the record movement buttons on the toolbar.
5. Close the South Seas Adventures application by double-clicking its system menu.
6. Close the Source Code Editor by double-clicking its system menu.

Using the QueryClose() Event Handler

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_ message in the application's 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 clicking its module branch in the Repository Explorer tree view.
2. Right-click on the module and select EditAllSource from the local pop-up menu.

3. Find the SSAWindow:FileExit() method:

```
METHOD FileExit() CLASS SSAWindow
    SELF:EndWindow()
    RETURN NIL
```

SSAWindow is the South Seas Adventures shell window. In an MDI application, double-clicking the shell window's system menu is equivalent 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.

```
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.

When a window's QueryClose() method returns TRUE, the window's Close() method is then invoked and the window is destroyed.

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.

Working with Icons and Cursors

This lesson introduces you to icons and cursors, and their uses in CA-Visual Objects 2.7. By the end of this lesson, you will know how to:

- Create icons and cursors
- Attach icons to your application windows
- Use icons as controls
- Customize your Repository Explorer with an icon
- Use cursors

Overview

Icons

As a user of Windows, you have seen icons used in the following situations:

- Icons used to represent programs and groups on your desktop. Typically, these icons are defined in the program's .EXE file, a .DLL file, or an .ICO file. The icons are identified to the desktop via the Program Properties.
- Icons within a program's Help About dialog box. In this case, the icon displays on a window as a Fixed Icon control.
- Icons used to represent a minimized window or application on the desktop.

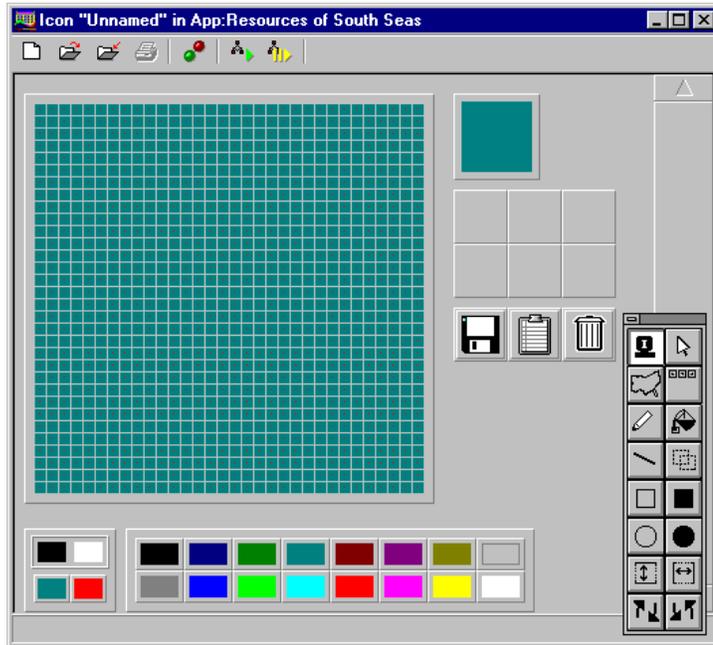
Creating an Icon

Let's start by creating a new icon to associate with the minimized Edit Customer window in the South Seas Adventures application:

1. Select the App:Resources module of the South Seas Adventures application in the Repository Explorer tree view.

2. Select the Image Editor command from the Tools menu.

The Image Editor window displays. The various parts of the Image Editor workspace, including the edit area, the color palette, the color indicators, and the tool palette, are shown below:

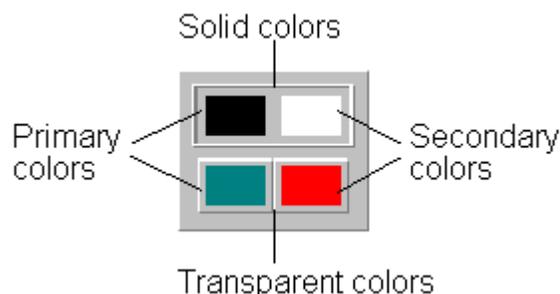


3. Maximize the Image Editor.

The Image Editor features many of the tools you may have seen in popular graphic editors and paint programs. These tools allow you to:

- Draw freehand
- Draw lines
- Draw rectangles (solid and outlined)
- Draw ellipses (solid and outlined)
- Fill areas
- Flip areas
- Rotate areas

When the Image Editor is first displayed, solid colors are selected by default, as indicated by the highlighted box in the color indicator:



The two left, or primary, colors in the indicator represent the colors used when you draw pressing the left mouse button. The two right, or secondary, colors in the indicator represent the colors used when you draw pressing the right mouse button.

The upper color indicators show *solid* colors. Solid colors appear on the icon exactly as they are displayed in your Image Editor.

NOTE: When being dragged, solid colors are only displayed in black and white.

The lower color indicators show *transparent* colors. These primary and secondary colors work differently from the solid colors:

- A primary transparent color will not appear on your icon when displayed, allowing what is behind the icon to show through. When selected, it fills the icon background completely—allowing you to view your icon in different color backgrounds.
- Areas that you draw using a secondary transparent color will be displayed as the complimentary color to what is behind it.

You can select Solid or Transparent colors from the Options menu.

Now, it's time to discover your hidden talents by drawing a simple icon, using solid colors. Since the icon will be used to represent customers for South Seas Adventures, let's draw a happy face:

1. Select the color yellow from the color palette by clicking the left mouse button.

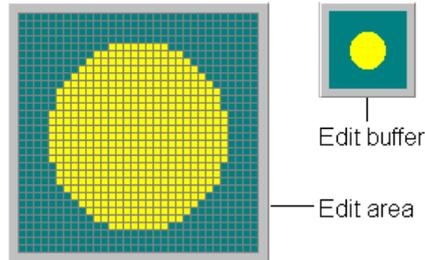
The left color in the indicator shows yellow as the primary color.

2. Select the Filled Ellipse tool from the Tools Palette, then move the mouse to the editing area.

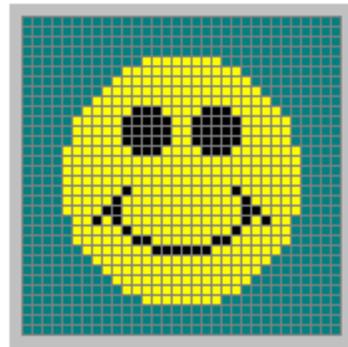


3. Click the left mouse button and drag it to create a circle.

Yellow circles appear in both the Edit Area and the Edit Buffer. The Edit Buffer displays the icon in the same size in which it will appear on the screen:



4. Select black as the secondary color by clicking with the **right** mouse button on the black color in the color palette.
5. Use the right mouse button to draw eyes and a mouth for the happy face. Use the Filled Ellipse tool for the eyes and draw the mouth using the Pencil tool:



Note: The green background that you see behind the happy face, which is a primary transparent color, will not show up when the icon displays.

Saving the Icon

Let's save your creation. Saving the icon will require a file name and an entity name. When referring to this icon in the application, you must use its entity name:



1. Select the Save Image File command from the File menu.

The Save Icon Entity and File dialog box displays.

2. Type **MYICON.ICO** as the file name.

For the directory path, select the `SAMPLES\SSATUTOR\FILES` subdirectory which is located below the CA-Visual Objects 2.7 installed directory (for example, `C:\CAVO27\SAMPLES\SSATUTOR\FILES`).

3. Type **MY_FIRST_ICON** in the Entity Name edit control.

An entity name may be long and descriptive. All characters are capitalized and spaces are replaced by underscores.

4. Choose OK to save the icon.

The Save operation creates an icon entity, an icon class and `Init()` method, a resource entity, and an `.ICO` file.

5. Close the Image Editor by double-clicking its system menu.

To view the four new entities, scroll down through the `App:Resources` entity list. There is an `Icon`, an `RCIcon`, a `Class`, and an `Init()` method for the class—all named `MY_FIRST_ICON`:



7. Your new `Icon` entity must be made visible to the rest of the application, so rebuild the application using the Build toolbar button.

Attaching Icons to Data Forms

In Windows applications, windows can be minimized. When a window is minimized, an icon displays in its place. In CA-Visual Objects 2.7, you can specify an icon for each window.

You are now going to attach your happy face icon to the Edit Customer window:

1. From the Repository Explorer, open the `Customer:Forms` module by clicking its branch.
2. Open the Window Editor by double-clicking the `EditCustomerWindow` window entity.
3. In the Data Window Properties window (under the `DataWindow` tab), select the `Large Icon` property.
6. Type **MY_FIRST_ICON** or click the down arrow button and select `MY_FIRST_ICON` from the drop-down list.
7. Select the `Small Icon` property.
8. Type **MY_FIRST_ICON** or click the down arrow button and select `MY_FIRST_ICON` from the drop-down list.
9. Save your changes by clicking the Save toolbar button.
10. Close the Window Editor by double-clicking its system menu.
11. Rebuild the application using the Build toolbar button.





To view the icon you specified, run the program as follows:

1. Click the Execute toolbar button.
2. Log in to the application (Name: **User**, Password: **Trainee**).
3. Open the Customer Browser by choosing the Open command from the File menu.

The Open File dialog box appears.

4. Click the Customer radio button, then choose OK.
5. Select any customer from the browser by clicking on a cell.
6. Click the Edit button to open the Edit Customer window.
7. Minimize the Edit Customer window to view its icon—the happy face.
8. Close the application by double-clicking its system menu and then answering Yes when prompted.

Labeling Your Application with an Icon

You can use an icon to uniquely identify the South Seas Adventures application on the Repository Explorer by following these steps:



1. Select the Application Properties toolbar button.

The Properties of South Seas dialog box displays.

2. Click the Explorer Icon button.

All of the icons defined in the application, as well as any in its path, will appear in the Icons of South Seas dialog box.

3. Click the happy face icon, then choose OK.
4. Choose OK to close the Properties of South Seas dialog box.
5. Return to the Repository Explorer. You will notice that the icon you just selected now displays on the South Seas Adventures application button.
6. Before proceeding, however, change the application icon back to the palm tree. Simply follow steps 1 through 4 above, and select the palm tree icon in step 3.

Icons in the Program Group

The icon that appears in the Repository Explorer is not always the program's icon in the program folder where the resulting .EXE file appears:



1. View the icon in the program group and create the executable file by choosing the Make EXE toolbar button.

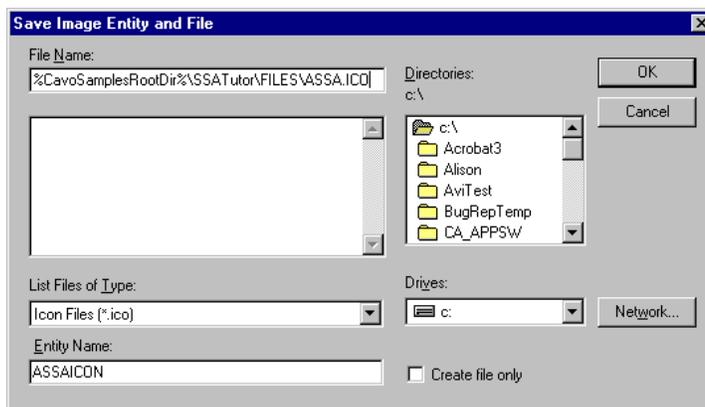
CA-Visual Objects adds the generated executable to the program folder specified in your application's properties where you see its icon. This is not the same icon as used in the repository. You will see this if you go into the Window's Explorer and locate SOUTHSEAS.EXE in your CAVO27\BIN directory.

2. Return to CA-Visual Objects.

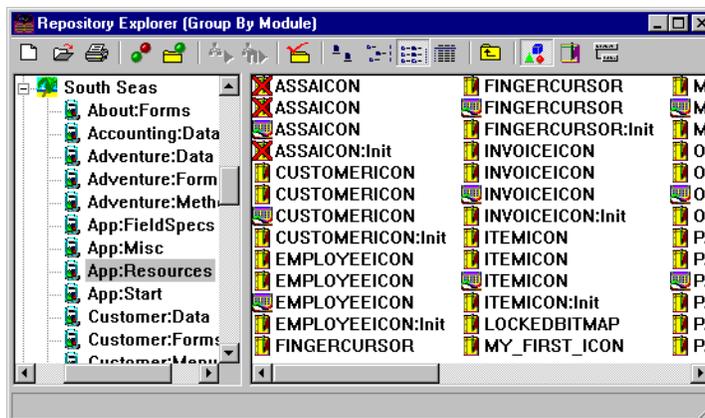
Change the Icon

To make the icon the same in both the Repository Explorer and program folder:

1. Select the App:Resources module, then open the SSAICON in the Icon Editor.
2. Select the Save Image File option from the File menu.
3. Enter an "A" at the beginning of both the File Name and the Entity Name so that it looks like this:



4. Close the Icon Editor so that you can see the entries in the Repository Explorer:



5. Rebuild the application using the Build toolbar button.



6. Create the executable file by choosing the Make EXE toolbar button.

7. You can now go back to the Windows Explorer to see the changed icon for the file.

The reason for this that Windows Explorer looks inside the .Exe file and takes the first Icon resource that it finds as its representation. Since the resources are added in alphabetical order, the “A” at the beginning of the name makes our new icon the first to be added.

Attaching Icons to Shell Forms

Just as you can minimize a window within an application, you can minimize the entire application. This capability is implemented by attaching a specific icon to the application’s main shell window.

Now, let’s attach an icon to the main shell window of the application:

1. From the Repository Explorer for the South Seas Adventures application, open the entity list for the SSA Shell:Forms module by clicking its branch.
2. Open the Window Editor by double-clicking the SSAWindow window entity.
3. In the Shell Window Properties window (under the ShellWindow tab), select the Small Icon property.
4. Click the down arrow button and select MY_FIRST_ICON from the drop-down list.
5. Notice that the Large Icon property is already set to SSAIcon.
6. Save your changes by choosing the Save toolbar button.
7. Close the Window Editor by double-clicking its system menu.



To view the icon you have specified, run the program as follows:

1. Rebuild the application using the Build toolbar button.
2. Click the Execute toolbar button to run the application.
3. After you log in to the South Seas Adventures application (Name: **User**, Password: **Trainee**), minimize the application.



The happy face icon is now used for the minimized shell window on the Windows taskbar. Recall that this is the Small Icon defines for the SSAWindow. Clicking the program item on the Windows taskbar will restore the application.

4. Now hold down the Alt key and press Tab once (keep the Alt key pressed). You will notice the palm tree icon (SSAIcon) in the Task window. Recall that this is the Large Icon defined for the SSAWindow. While still holding down the Alt key, press Tab until the palm tree is selected again. Then release both the Alt and Tab keys.
5. Close the South Seas Adventures application by double-clicking its system menu and answering Yes when prompted.

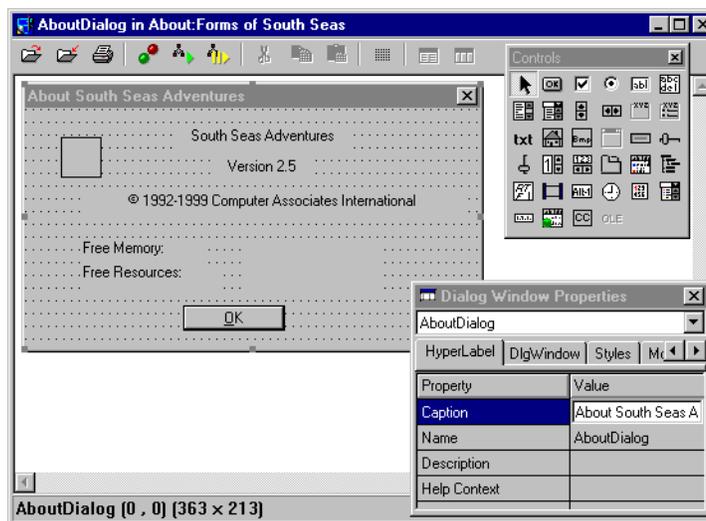
Displaying an Icon on a Window

You can display any icon in a window as a control. An Icon entity name must be provided as the Caption property for a Fixed Icon control. For additional information on controls, see [Chapter 5: Creating and Using Windows](#) in this guide.

Let's replace the icon on the About Help dialog form:

1. From the Repository Explorer open the About:Forms module by clicking its branch.
2. Select the AboutDialog window from the entity list by double-clicking it.

The Window Editor appears:



3. Click the fixed icon control on the dialog form.
4. Select the Caption property from the Properties window (under the HyperLabel tab).

It is currently set to SSAICON, which is the name for the palm tree icon you have seen used throughout the South Seas Adventures application.

5. Enter **MY_FIRST_ICON** to change the icon to the happy face you created earlier in this exercise.

6. Close the Window Editor by double-clicking its system menu and answering Yes when prompted to save the changes.

To view the icon you specified, run the program as follows:



1. Rebuild the application by clicking the Build toolbar button.
2. Run the application by clicking the Execute toolbar button.
3. Log in to the application (Name: **User**, Password: **Trainee**).
4. Select the About command from the Help menu.

Notice the happy face icon displays in the dialog window.

5. Close the application by double-clicking its system menu and answering Yes when prompted.

Using Predefined Cursors

Typically, when an application is processing, the hourglass pointer or *cursor* appears. CA-Visual Objects provides this and other predefined pointers. To use them, simply assign your window's Pointer property with the appropriate Pointer object, as in:

```
SELF:Pointer := Pointer{POINTERHOURLASS}
```

Where SELF refers to the current window object.

Creating and Modifying Cursors

You can designate a mouse cursor, or pointer, for a shell window or dialog by using its Mouse Pointer property in the Window Editor. You may also modify a mouse cursor by directly modifying the source code, as demonstrated here.

To define a mouse cursor of your own for a window, let's revisit the Image Editor:

1. Select the App:Resources module of the South Seas Adventures application by clicking its branch on the Repository Explorer tree view.
2. To create a new cursor, select the Image Editor command from the Tools menu or click the New Entity toolbar button.

When you first enter the Image Editor, it is in Icon Mode so you must switch to Cursor Mode.



3. To switch to Cursor Mode, click the Cursor Mode button from the tool palette.

Notice that the top-left pixel in the edit area is marked with a dotted line around it. This indicates the *hot spot* of the cursor. When using a cursor, the hot spot is the actual location of the cursor on the screen. For example, the hot spot for the standard arrow pointer is the tip of the arrow.

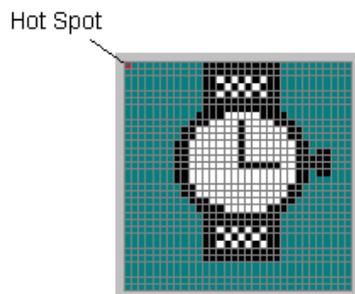
Let's import a cursor that was installed with the tutorial:

1. Select the Open Image File command from the File menu.

The Open Cursor File dialog box appears.

2. Select the WATCH.CUR file in the SAMPLES\SSATUTOR\FILES subdirectory below the CA-Visual Objects 2.7 installed directory (for example, C:\CAVO27\SAMPLES\SSATUTOR\FILES\WATCH.CUR) and choose OK.

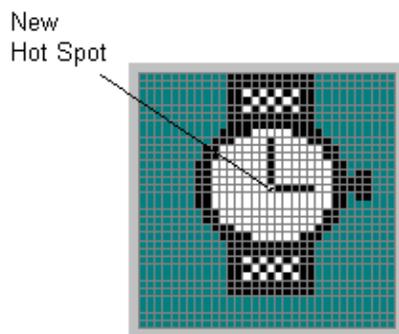
You are going to use this new watch cursor in place of the one that is currently being used in your code. The hot spot for this cursor was mistakenly left in the top-left corner:



To correct the hot spot:

1. Move the mouse pointer to the intersection point of the hour and minute hands.
2. While pressing the Ctrl key, click the left mouse button.

This places the hot spot at the pointer location:



The hot spot is now in the center of the watch.

3. To save the new cursor definition, select the Save Image File command from the File menu.

The Save Cursor Entity and File dialog box appears. As with saving icons, you must specify a file name and an entity name.

4. Type **NEWWATCH.CUR** in the File Name edit control and **WATCH_CURSOR** in the Entity Name edit control.

The entity name is used when referring to the cursor within your code.

5. Choose OK.
6. Close the Image Editor by double-clicking its system menu.

The `SSAWindow:OptionsReindex()` method currently uses the standard hourglass cursor to indicate that processing is taking place. You are going to change this source code to use your watch cursor instead:

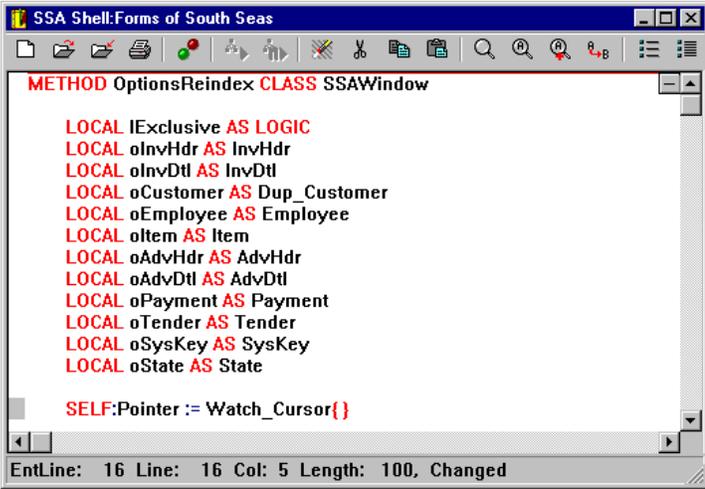
1. Open the SSA Shell:Forms module by clicking its branch on the Repository Explorer tree view.
2. Open the Source Code Editor by double-clicking the `SSAWindow:OptionsReindex()` method.
3. At or about line 16 of the method (just after the declarations), replace the following line:

```
SELF:Pointer := Pointer{POINTERHOURLASS}
```

With:

```
SELF:Pointer := Watch_Cursor{}
```

Your method now looks like this:



```

METHOD OptionsReindex CLASS SSAWindow

LOCAL IExclusive AS LOGIC
LOCAL oInvHdr AS InvHdr
LOCAL oInvDtl AS InvDtl
LOCAL oCustomer AS Dup_Customer
LOCAL oEmployee AS Employee
LOCAL oItem AS Item
LOCAL oAdvHdr AS AdvHdr
LOCAL oAdvDtl AS AdvDtl
LOCAL oPayment AS Payment
LOCAL oTender AS Tender
LOCAL oSysKey AS SysKey
LOCAL oState AS State

SELF:Pointer := Watch_Cursor{}

```

EntLine: 16 Line: 16 Col: 5 Length: 100, Changed

4. Save your changes by choosing the Save command from the File menu.
5. Close the Source Code Editor by double-clicking its system menu.
6. Rebuild the application using the Build toolbar button.
7. To run the program, click the Execute toolbar button.



8. Log in to the application (Name: **User**, Password: **Trainee**).
9. Select the Re-index Database command from the Options menu.
Your watch cursor is now visible for the duration of the process.
10. Exit the application by double-clicking its system menu and answering Yes when prompted.

Summary

In this lesson, you learned how to create icons and cursors. You:

- Used icons to depict minimized windows
- Placed an icon in a fixed control
- Customized the Repository Explorer branch using a specialized icon
- Defined the icon to use for the generated .EXE file in the Windows folder
- Modified a cursor by redefining its hot spot and used this cursor as an indicator for a particular process in the application

In the following lesson, using the DrawObject classes, you will learn how to create, resize, and modify bitmap and text objects.

Working with Draw Objects

In this lesson, you will learn how to display and manipulate bitmap and text objects using the DrawObject classes. You will also see when it is appropriate to display these objects.

Overview

The opening dialog box of the South Seas Adventures application contains a bitmap and text, which were implemented using objects from the DrawObject hierarchy. This dialog box, shown below, is used as an example in the exercise that follows:



The opening dialog box will be given the capability to resize its contents based on its own size. This means, that as the dialog box grows or shrinks, any text and pictures within it grow or shrink proportionately.

These are the assumptions that were made prior to creating the dialog box:

Entity	Assumption
Window	The window is divided into two equal parts. The left side is used to display the bitmap. The right side is used to display the text and an OK push button.
Text	The text is to be centered on the right side of the window.
Text width and height	The right side of the window can hold a maximum number of lines (height) and a maximum number of characters (width) defined by the constants, <code>LINES_DOWN</code> and <code>CHARS_ACROSS</code> , respectively.
OK button	The OK push button is to be centered at the bottom of the right side of the window. Its size remains constant and is used to determine the minimum height and width of the window, since this button should always remain visible.

Exercise

In the following exercise, you have the opportunity to examine the source code in the South Seas Adventures application responsible for creating and displaying the opening dialog box. During this exercise, you will:

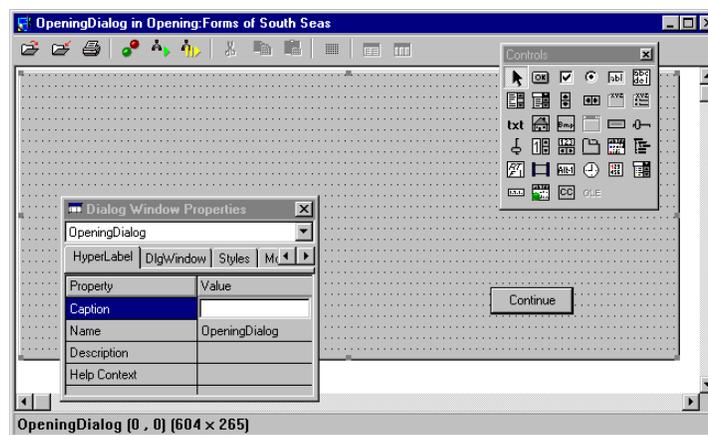
- Modify the `OpeningDialog` window entity to allow the opening dialog box to be resized
- Examine the source code necessary for resizing
- Examine the source code for displaying and dynamically resizing a bitmap
- Examine the source code for displaying and dynamically resizing text
- Examine the source code for dynamically positioning the OK push button
- Run the application to see the results of the generated source code

Making the Dialog Box Resizable

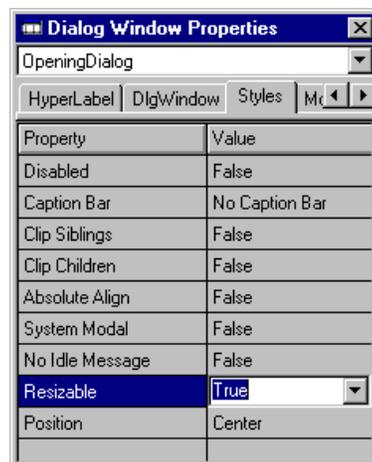
Initially, the opening dialog box was set up to be a fixed size; therefore, the first thing you will do in this lesson is make it resizable. The ability to resize a dialog box is controlled by one of its style properties. To change it, open the dialog box in the Window Editor as follows:

1. From the South Seas Adventures Application branch click the Opening:Forms module branch.
2. Double-click the OpeningDialog window entity in the Repository Explorer list view.

The Window Editor, which was used to create the window and the OK button, is displayed:



3. Select the Dialog Window Properties, Styles tab.
4. Click the Resizable property and choose True from the drop-down list box:



This enables you to resize the South Seas Adventures application opening dialog box.



5. Select the Save toolbar button.

Exit the Window Editor by double-clicking its system menu.

The Resize Event

When you make a window resizable, as described in the previous steps, certain aspects of the physical resizing event are handled automatically. When the user attempts to resize the window, the system generates a Resize event that is, by default, handled by the `Window:Resize()` event handler method. However, if you have any special processing unique to your window, you must code a `Resize()` method of your own to handle it. (This has already been done for the `OpeningDialog` class.)

To see how the Resize event is handled for the opening dialog box, find the `OpeningDialog:Resize()` method in the entity list and double-clicking it.

The following code is loaded in the Source Code Editor:

```
METHOD Resize(oRSE) CLASS OpeningDialog
LOCAL iMinHeight, iMinWidth AS INT
SUPER:Resize(oRSE)
// Put your changes here
// Screen size is based on size of the button
iMinHeight := SELF:oCCOkButton:Size:Height * 5
iMinWidth := SELF:oCCOkButton:Size:Width * 3
IF SELF:Size:Height < iMinHeight .OR. ;
  SELF:Size:Width < iMinWidth
  // Don't let screen get too small
  SELF:Size := Dimension{MAX(iMinWidth, ;
    SELF:Size:Width), Max(iMinHeight, ;
    SELF:Size:Height)}
ENDIF
// Repaint on every size since entire
// screen is proportional

SELF:Repaint()
RETURN NIL
```

The code checks the height and width of the window and, if necessary, resets it. To make sure that the OK push button is always visible in the dialog box, the size of the button was taken into consideration in the calculation:

```
iMinHeight := SELF:oCCOkButton:Size:Height * 5
iMinWidth := SELF:oCCOkButton:Size:Width * 3
```

From this calculation, you can see that the minimum height of the window is five times the height of the button. Similarly, the width of the window is set to three times the width of the button.

The following line forces the window to repaint each time the window is resized:

```
SELF:Repaint()
```

This forces an Expose event to occur. The Expose() event handler can then repaint our objects in the window, which is discussed in more detail later in this lesson.

Now that you have explored the Resize() method, it is time to move on to the code in which the bitmap is displayed in this dialog box. Before moving on, close the Source Code Editor by double-clicking its system menu.

Using Bitmaps

The opening dialog box for the South Seas Adventures application displays a bitmap image from a .BMP file.

To display a bitmap from a .BMP file, you must:

- Declare the file as a resource
- Create a BitmapObject (a subclass of DrawObject)
- Create an Init() method for the BitmapObject

Declaring a .BMP File as a Resource

A resource declaration statement is needed for accelerators, bitmaps, cursors, dialogs, icons, and menus. The code for these is usually generated by the associated visual editor. However, on occasion you must directly enter the code for the resource declaration via a RESOURCE statement in the Source Code Editor. Such is the case with bitmap images.

Note: When the application is built, all RESOURCE statements are sent directly to the Windows resource compiler.

To view the source code responsible for declaring the bitmap used in the opening dialog box:

1. Select the App:Resources module by clicking its module branch.
2. Right-click the App:Resources branch and choose Edit All Source from the local pop-up menu.
3. Find the RESOURCE SSABitmap entity, which reads as follows (assuming you have installed CA-Visual Objects 2.7 to C:\CAVO27):

```
RESOURCE SSABitmap Bitmap;  
    c:\cavo27\samples\ssatutor\files\ssabmp
```

4. Exit from the Source Code Editor by double-clicking its system menu.

Tip: Use either the Find toolbar button or the Go to Entity toolbar button in the Source Code Editor to locate an entity quickly.

Creating a Bitmap Object

The next step is to create an object of the Bitmap class that refers directly to the resource declared in the previous steps. This has already been done in the South Seas Adventures application, but it is helpful to look at the source code to understand the connections.

Declaring a Bitmap
Subclass: SSABitmap

At this point, the App:Resources Entity Browser should still be open:

1. Locate the SSABitmap class entity, and double-clicking it to load it into the Source Code Editor.

You see the following line of code:

```
CLASS SSABitmap INHERIT Bitmap
```

2. Return to the Repository Explorer by clicking the Window menu and select Repository Explorer.
3. Locate the SSABitmap:Init() method and double-clicking it.
4. Scroll through the Source Code Editor window to view the source code for both entities, which should look as follows:

```
CLASS SSABitmap INHERIT Bitmap
METHOD Init() CLASS SSABitmap
    SUPER: Init(ResourceID{"SSABitmap", _GetInst()})
RETURN SELF
```

This code is fairly straightforward. First, we have created a subclass of the Bitmap class. Then, we have defined an Init() method to be executed when objects of this subclass are instantiated.

Within the Init() method, note the use of the ResourceID class, which provides a unique identifier for a resource based on its name. You could declare your resources using unique identifiers rather than names, but working with resource names and converting them using the ResourceID class is much easier.

5. When you are finished looking at this source code, close the Source Code Editor by double-clicking its system menu.

Instantiating an
SSABitmap Object

After defining the class and instantiation code for the new bitmap, it is necessary to give the opening dialog box access to the bitmap. The most logical place to do this is from within the OpeningDialog:Init() method:

1. Close the App:Resources entity, and open the Opening:Forms module by clicking its module branch.

- Find the `OpeningDialog` class entity and double-click it to view it in the Source Code Editor.

You see the following code:

```
CLASS OpeningDialog INHERIT DIALOGWINDOW
  PROTECT oCCKButton AS PUSHBUTTON
  //{{%UC%}}
  //USER CODE STARTS HERE (do NOT remove this line)
  PROTECT LogoBitmap AS SSABitmap
```

- Return to the Repository Explorer by clicking the Window menu and select Repository Explorer.
- Find the `OpeningDialog:PostInit()` method and double-clicking it.
- Scroll through the Source Code Editor window to view the source code for both entities. The code should look as follows:

```
METHOD PostInit( oParent, uExtra ) CLASS:
  OpeningDialog
  SELF:LogoBitmap := SSABitmap{ }

  RETURN NIL
```

In the `OpeningDialog:PostInit()` method, the *LogoBitmap* instance variable (declared in `CLASS OpeningDialog`) is set to an `SSABitmap` object. Creating the bitmap in this manner allows you to create the object once, as part of the window instantiation, and then reuse it each time the window is redrawn.

- When you are finished looking at this source code, close the Source Code Editor by double-clicking its system menu.

Drawing a Bitmap on a Window

Now, let's look at the code in which the bitmap is actually drawn on the dialog box. Earlier, when we discussed resizing a window via the `Resize` event, we mentioned the `Expose` event that was triggered as a result of calling the `Window:Repaint()` method. The `Expose` event can now be viewed in greater detail.

The Expose Event

An `Expose` event occurs whenever the windows need repainting. This can occur under any of the following circumstances:

- The window is first shown
- The window is partially uncovered by another window
- The window changes in size
- The window is being restored after being minimized
- A call is made to the window's `Repaint()` or `RepaintBoundingBox()` method

At this point, the `Opening:Forms Entity Browser` should still be open. Find the `OpeningDialog:Expose()` method and double-click it to load it into the Source Code Editor.

The following lines of code define the available size (adjusted to remove four pixels for each border) where the available width for the bitmap is the integer `iMidWidth`:

```
// Get screen height and width minus borders
iHeight:= SELF:Size:Height-8
iWidth:=SELF:Size:Width-8
iMidWidth:=INT(iWidth/2)
```

Now let's examine the `SELF:Draw()` line of code (located around line 16) which is responsible for drawing the `SSABitmap`:

```
// Draw bitmap sized and positioned
// relative to window
SELF:Draw(BitmapObject{Point{2, 2},;
    Dimension{iMidWidth - 4, iHeight - 4}, ;
    LogoBitmap})
```

This is a fairly complicated line of code. Let's examine its individual components to get a better understanding of what is going on:

Component	Definition
<code>BitmapObject{...}</code>	Creates a bitmap draw object.
<code>Point{2, 2}</code>	Defines the point, in pixels, at which to start drawing the bitmap object.
<code>Dimension{iMidWidth-4, iHeight-4}</code>	Defines the pixel width and height of the object to be drawn. In this case, the width is half the window width less four pixels, and the height is the window height less four pixels. This ensures that the size of the bitmap is always relative to the size of the window.
<code>LogoBitmap</code>	The bitmap object that was assigned when the opening dialog box was created.
<code>SELF:Draw(...)</code>	Draws the logo on the dialog box.

Important! You should always use the `Expose()` method when displaying `DrawObject` class entities. Also, use the `Window:Draw()` method to display your individual `DrawObject`. Never use the `DrawObject:Draw()` method directly, as it is called by the `Window:Draw()` method.

Using Text Objects

The `Expose()` event handler method also has code for displaying the text on the dialog box. This is accomplished using the `TextObject` class, which is like `BitmapObject` in that it is a subclass of `DrawObject`.

You should still have the source code on your window at this point. Just below the line of code discussed previously for drawing the bitmap, you should see the following lines of code:

```
iFontWidth := INT(iMidWidth / CHARS_ACROSS)
iFontHeight := INT(iHeight / LINES_DOWN)
```

These lines calculate the width and height of the font, in canvas coordinates, used to display the text. The width is calculated as the window width divided by the number of characters allowed (CHARS_ACROSS). The height is calculated as the window height divided by the number of lines allowable (LINES_DOWN). In this way, the font is scaled based on the size of the window.

Note: Both CHARS_ACROSS (set to 30) and LINES_DOWN (set to 7) are defined as constants in this same module. Alternatively, you can view them in the Source Code Editor by double-clicking the corresponding entity.

Next, the font is instantiated using the height and width calculated previously:

```
oTextFont := Font{FONTROMAN, ;
    Dimension{iFontWidth, iFontHeight}}
```

Finally, the TextObject object is created and drawn on the dialog box. The code for doing this is similar to the code for drawing the bitmap that you saw earlier. The TextObject object is instantiated within the call to SELF:Draw(), and the starting point is determined dynamically, based on the font and window size:

```
sLine1 := "South Seas Adventures"
SELF:Draw(TextObject{Point{iMidWidth + ;
    Int((CHARS_ACROSS-Len(sLine1))/2) * iFontWidth, ;
    Int(Float(iHeight) * .5)}, sLine1, oTextFont, oBlack})
```

Dynamic Positioning of Controls

The OK push button is the only entity on the window that is not scaled—although it could be. Since this is the only push button control on the window, it is best to fix its size as you could not afford to lose it to resizing.

This button, however, is dynamically positioned so that it is always visible. The positioning of the OK button is also handled within the Expose() event handler. This should still be on your window. Look immediately below the line of code discussed previously for drawing the text. You see the following lines of code:

```
// Position push button
oCCOkButton:Origin := Point{ ;
    Int((Float(iWidth) * .75) - ;
    (oCCOkButton:Size:Width / 2)), 10}
```

In this code, *oCCOkButton* is the name of the push button object. By calculating its *Origin* property based on the current width of the dialog box, the position of the OK push button is computed at runtime each time the dialog box is redisplayed.

Viewing the Results in the Application

This concludes the overview of the source code used to control the opening dialog box. If you like, you can verify and examine the results in the South Seas Adventures application:



1. Close the Source Code Editor by double-clicking its system menu.
2. Even though you have not made any source code changes, you need to rebuild the application, using the Build toolbar button, because of the minor change you made earlier in the Window Editor.
3. Run the South Seas Adventures application by clicking the Execute toolbar button.
4. Resize the dialog box by dragging the right border to the left. Note that everything except the push button is scaled down.
5. Continue to resize the dialog box to see the results of the source code you have been reviewing. When you are done verifying the results, choose OK.
6. At the Login dialog box, choose the Cancel button.

Summary

In this lesson, you learned how to use the `BitmapObject` and `TextObject` classes, and how to scale objects created by these classes according to the window in which they reside. You also learned about the `Expose` and `Resize` events.

In the following lesson, you will learn how to use the CA-Visual Objects Report Editor to define, customize, preview, and print a report.

Reporting with the Report Editor

In this lesson, you will learn how to create and manipulate reports using the Report Editor.

The Report Editor is a powerful reporting facility that offers ease of use through its intuitive user interface. Using the Report Editor, you will learn how to define, customize, preview, and print a report.

Exercise

During this exercise, you are going to define the Customer List report. As you develop this report, you will:

- Gain a working knowledge of the Report Editor's menus and toolbars
- Learn how to create report headers and footers
- Provide your own titles and pictures
- Insert, delete, and sort report data fields
- Use the Report Editor help facility
- Learn how to pass parameters to your Report Editor report

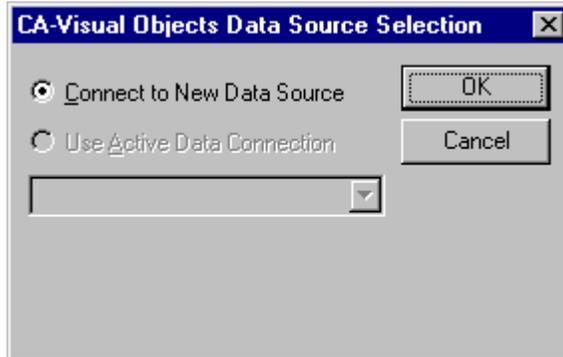
Using the Report Editor

In this exercise, you will create a report that works on any user's machine, independent of the installation drive and directory. This is accomplished by using data servers that do not contain any path information for data and index files. For more information, refer to [Appendix A: Creating a Path-Independent Application](#).

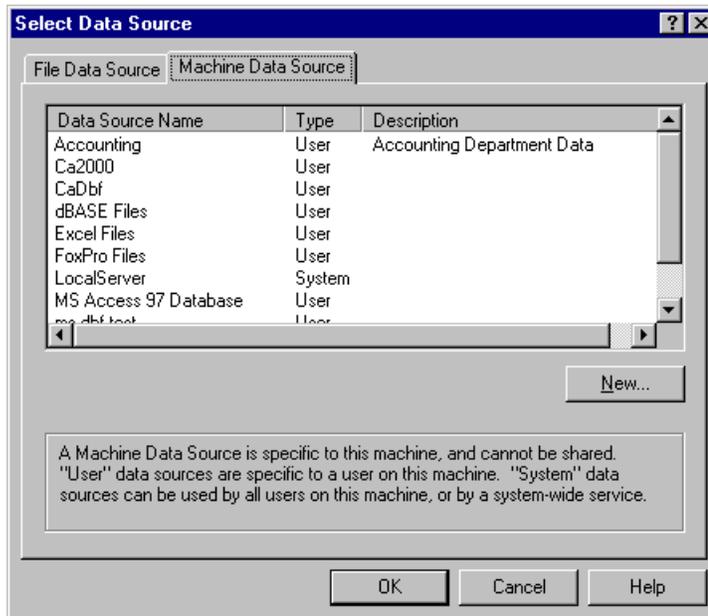
1. Open the South Seas Adventures Application by double-clicking its branch on the Repository Explorer.
2. Select the Customer:Reports module by clicking its branch in the Repository Explorer tree view.
3. Choose the Open Entity toolbar button, and select Report Editor from the local pop-up menu that appears.



The CA-Visual Objects Data Source Selection dialog box appears:

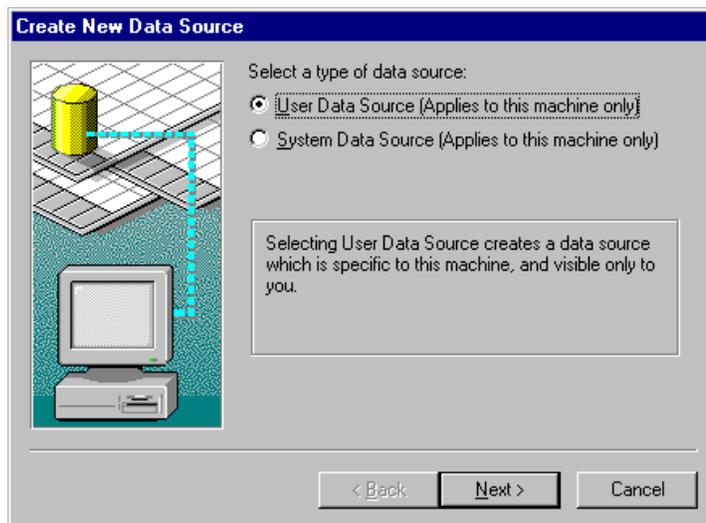


4. Select the Connect to New Data Source radio button and click the OK button.
5. Change to the Machine Data Source tab:



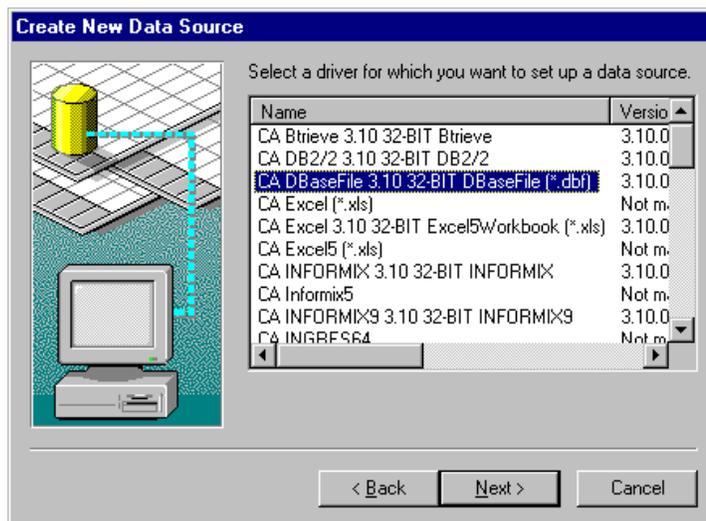
- Click the New button to create a new data source.

The Create New Data Source dialog appears:



Select the User Data Source radio button and click Next.

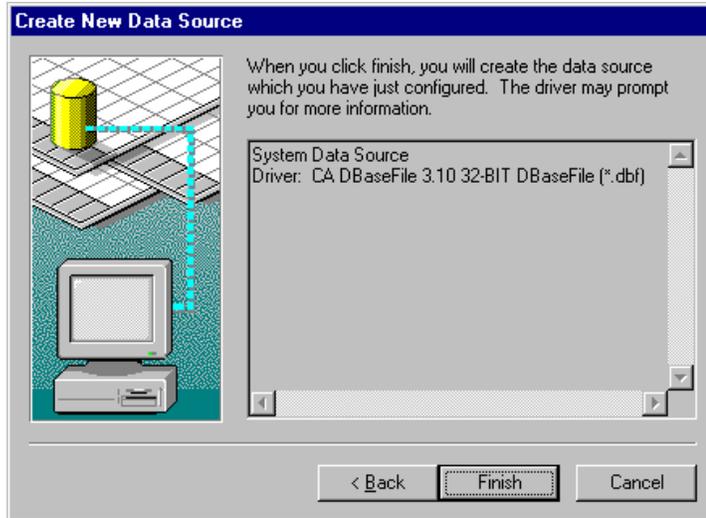
A list of installed drivers appears:



- Scroll through the Installed ODBC Drivers list box and select CA DbaseFile 3.10 32-BIT DBaseFile (*.dbf).

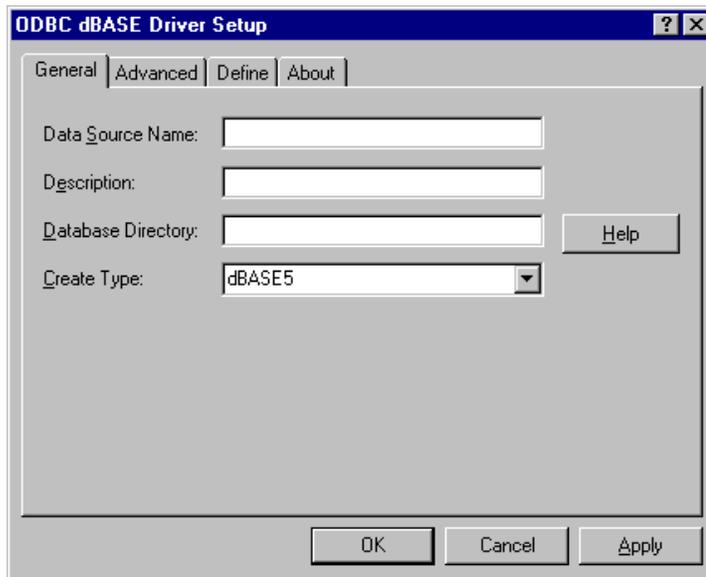
8. Select the Next button to use this data source.

At this point we have selected the driver to be used with our report and a new window will display the options that you have chosen:



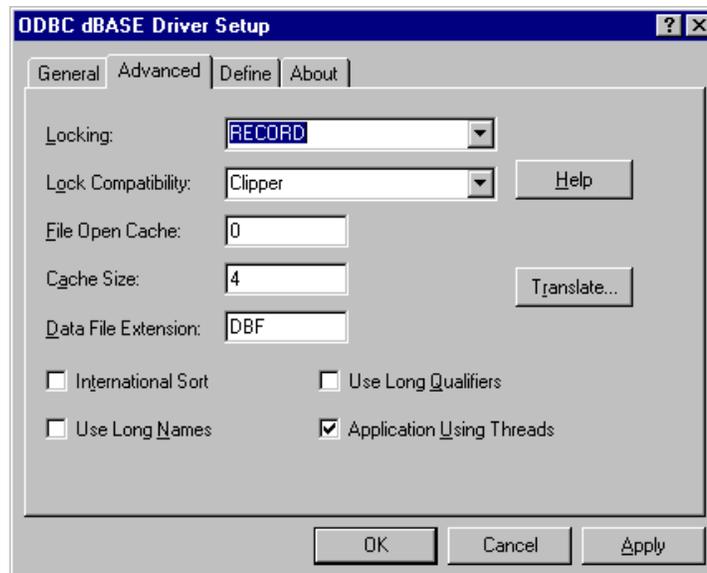
Press the Finish button to accept these options.

The ODBC dBASE Driver Setup dialog box appears:



9. In the Data Source Name edit control, type **CA-Visual Objects 2.7 Xbase**.
10. In the Description edit control, type **CA-Visual Objects 2.7 Report Editor**.
11. In the Database Directory edit control, type in the path to the SAMPLES\SSATUTOR subdirectory, which is located in the CA-Visual Objects 2.7 installed directory. For example, C:\CAVO27\SAMPLES\SSATUTOR.

12. From the Create Type drop-down list box, select Clipper.
13. Select the Advanced tab:



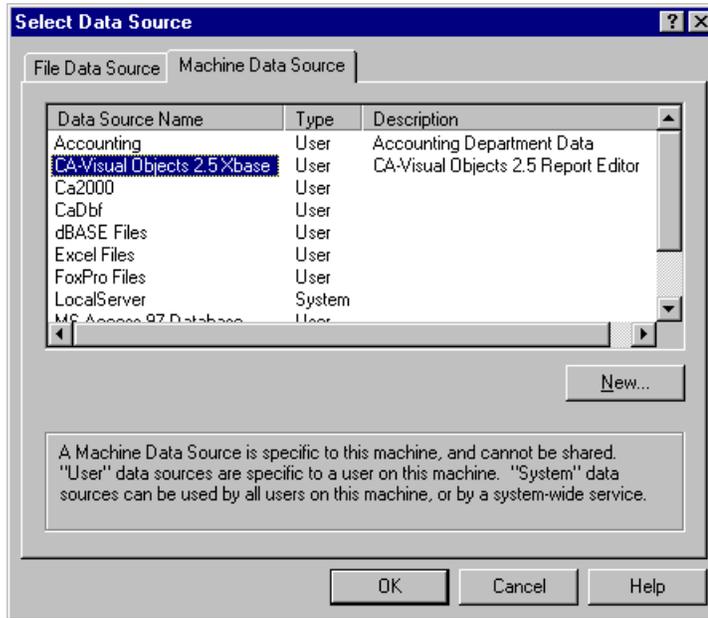
Choose RECORD locking from the Locking drop-down list box.

14. Each Xbase product uses its own style of locking, therefore, choose Clipper from the Lock Compatibility drop-down list box.

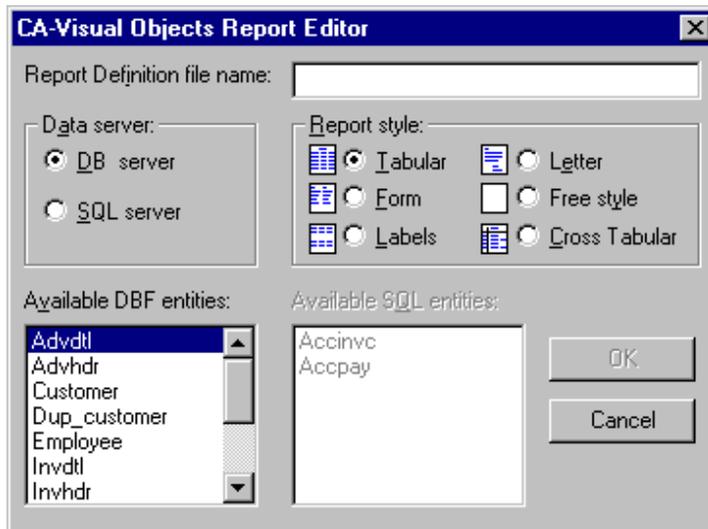
You are now finished defining the data source.

Choose OK.

- You are returned to the Select Data Sources dialog box. Select CA-Visual Objects 2.7 Xbase (which we just created) and click the OK button:



The Report Editor dialog box is displayed:



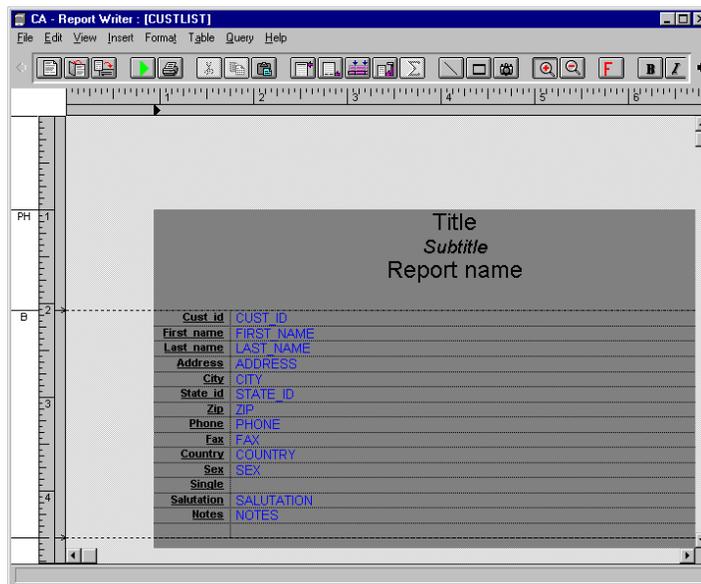
- Type **CustList** in the Report Definition File Name edit control.
The name you enter here is used by the Report Editor to generate the CustList class that is used to run the report from your application.
- The CustList report will use data from the CUSTOMER.DBF file, for which we have already defined a data server (see [Chapter 3: Working with Data Servers](#)). Therefore, select Customer from the Available DBF Entities list box.

18. Because the Available DBF Entities list box allows multiple selections, you must also deselect Advdtl.
19. Click the Form radio button in the Report Style group box.

When you create a new report, the Report Editor allows you to choose from various default report styles. Based on the style that you select, the Report Editor generates a default report.

20. Choose OK.

The Report Editor will now be launched. With just a few selections, you have a predefined report—with titles, labels, and data:



21. Preview the report by selecting Print Preview from the File menu.
22. When you are finished previewing the report, select Close from the File menu.

A Quick Tour

At first, the Report Editor may remind you of your favorite word processor. That's because it uses much of the same functionality you have seen in word processors. If you want text on the report, just type it in. To format your report, you will use the Report Editor's font toolbar and also rely heavily on tables, both of which are available in most word processors.

The main report editing area is known as the Report Definition. Two rulers, one along each axis of the report page, allow you to see where you are in your report at all times.

The *Section Name* window, along the left side of the report definition area, displays letters identifying each section of your report: PH (Page Header), B (Body), and PF (Page Footer).

Adding Your Personal Touch

The Report Editor has created a page header with the default text “Title,” “Subtitle,” and “Report Name.” Let’s customize this report header to better meet our needs:

1. Change the title by highlighting Title and typing **Customer List**.
2. Change the subtitle by highlighting Subtitle and typing **by Customer #**.
3. Highlight Report Name, and type **as of** with a space character after it.
You will next add a field to this line so that it displays the current date.

Adding Fields

The Report Editor allows you to add fields to your report. Fields can be the following types: database, computed, parameter, system-defined, or user-defined.

Next, add the system date as part of the header:

1. If your cursor is not already positioned after the “as of” text that you just entered, position it there.
2. Choose Field from the Insert menu.

The Insert Field dialog box appears:

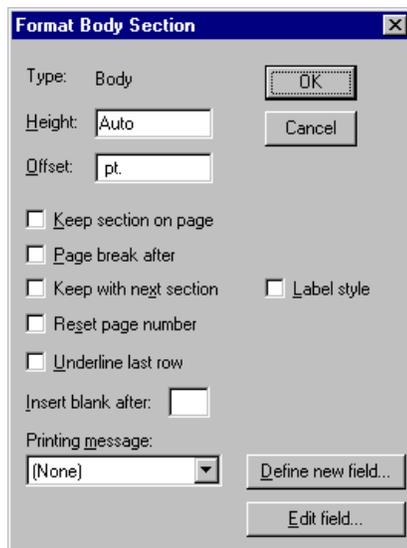


- From the Fields list box, select `_date` and choose Insert.

The `_date` field is a system-defined computed field that returns the current date. It now appears as a place holder in your report name (as of `_date`).

Click at any point on the body section of the report (anything below the header section), and then select the Section command from the Format menu.

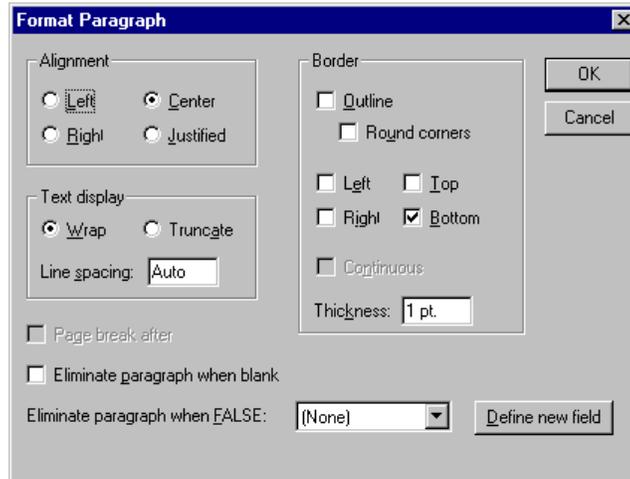
The Format Body Section dialog box appears:



- Select the Keep Section on Page check box in order to prevent the section from being split across two pages, and then choose OK.

Now, let's format the page header and footer:

1. Click the last line of the page header section, and then select the Paragraph command from the Format menu.
2. To underline each page header, select the Bottom check box in the Border group box, and then choose OK:



Note: Each line in the page header section has a distinct paragraph setting. The change that you just made will apply only to the last line in the header, since that line was selected when you chose the Format Paragraph command. To apply the same paragraph setting to all lines in the header, highlight all the lines before choosing the Format Paragraph command.

The next few steps place the company information in the page footer.

3. Place your cursor on the first line of the Page Footer section and type the following, pressing Enter after each line:

South Seas Inc.
1234 Seashore Drive
Montego Bay, Jamaica

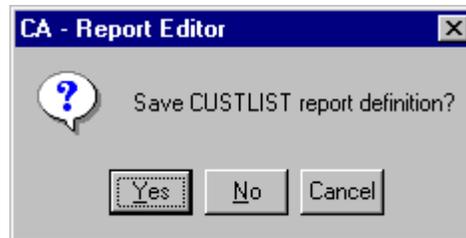
4. Double-click the first line of the Page Footer.
The Format Paragraph dialog box appears again.
5. To draw a line above each page footer, check the Top check box in the Border group box, and then choose OK.
6. By now, you probably want to see what your report looks like, so select Print Preview from the File menu.
7. Once you are done previewing and/or printing your report, choose Close from the file menu to return to the Report Editor.

Saving Your Work

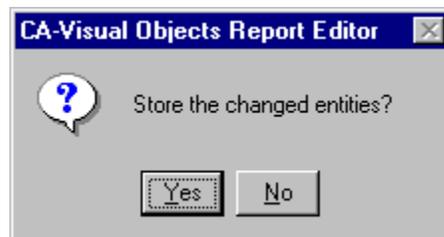
Save your report by:

1. Choosing save from the File menu. If you do not save the report before exiting, Report Editor will remind you to save your changes.
2. Close the Report Editor by selecting the Close command from the File menu.

The Report Editor prompts you to store the changed entities, as follows:



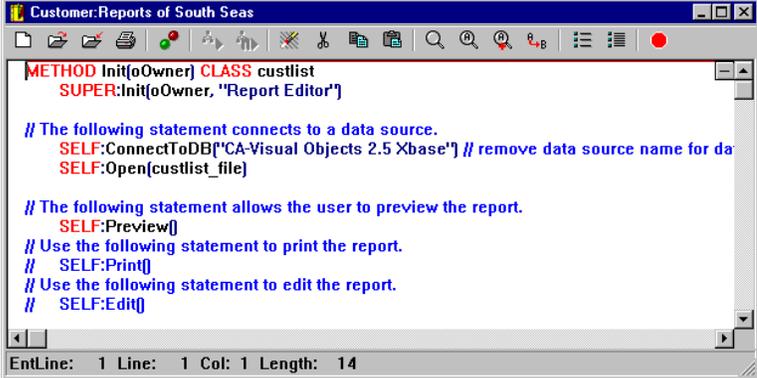
3. Choose Yes to save the changed entities.
4. At this point the repository questions whether to save the entities:



5. Choose Yes to save the changed entities.
6. The Report Editor then generates the CustList class in the Customer:Reports module.

Running Your Report Within Your Application

To run the report from within your application, CA-Visual Objects generates the following code to display your report in Preview mode:



```
METHOD Init(oOwner) CLASS custlist
SUPER:Init(oOwner, "Report Editor")

// The following statement connects to a data source.
SELF:ConnectToDB["CA-Visual Objects 2.5 Xbase"] // remove data source name for da
SELF:Open(custlist_file)

// The following statement allows the user to preview the report.
SELF:Preview()
// Use the following statement to print the report.
// SELF:Print()
// Use the following statement to edit the report.
// SELF>Edit()

EntLine: 1 Line: 1 Col: 1 Length: 14
```

Note: You can also use CustList as a menu item's event name or as the name of a push button. When the menu item or push button is selected, the report will run.

For the purposes of this report, however, you will be passing a parameter later in this exercise. To accommodate the use of some special report preview dialog boxes, you need to comment out the following line:

```
SELF:Preview( )
```

Use the following these steps:

1. Double-click the CustList:Init() method from the Customer:Reports entity list to bring up the Source Code Editor:
2. Move the cursor to the line of code reading SELF:Preview(), and insert the characters // at the beginning of the line.
3. Close the Source Code Editor by double-clicking its system menu, and answering Yes when prompted to save your changes.

Report Parameters

In the previous section, you were briefed on running your report within your application. Often, it is necessary to *filter* the report based on information at runtime, for example, printing a Payment Report for the month of June or printing Invoice number "04729."

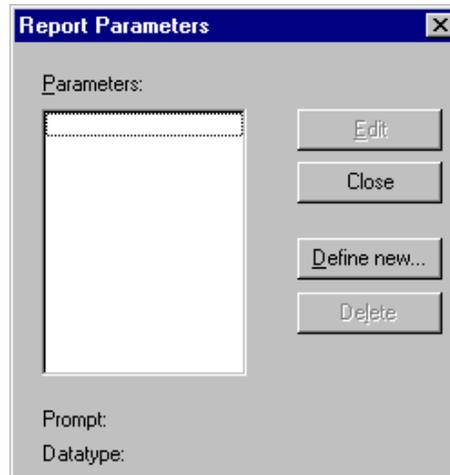
In the South Seas Adventures application, the Customer List report should be able to print either all customers or a single customer. To do this, you can define parameters for the report. These parameters can then be inserted into the query for the report.

First, let's define a parameter for the report:

1. To reopen your report, double-click the CustList report entity.
This will bring you back into the Report Editor.

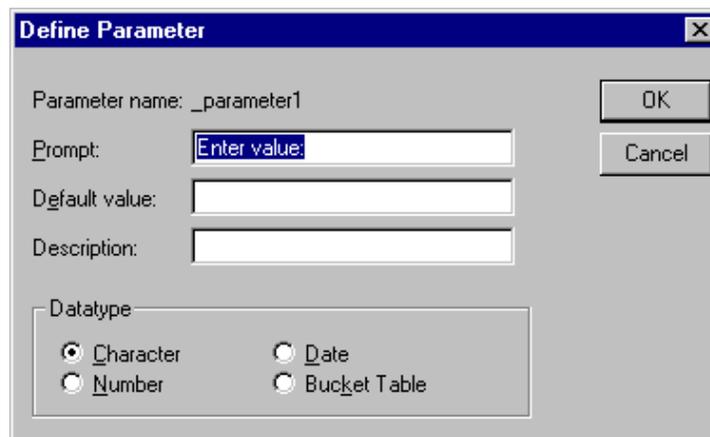
2. Select the Report Parameters command from the Edit menu.

The Report Parameters dialog box appears:



3. Choose the Define New button.

The Define Parameter dialog box appears:



When you invoke a report that expects parameters, the application may either have the parameters hard coded in the program or allow the user to enter them at runtime. The Prompt edit control determines what the end user will see if the application does not pass the required parameters.

4. For this report, type **Enter a customer #:** in the Prompt edit control.
Note: The Default Value edit control determines the default to use if the end user does not type anything.
5. In the Description edit control, type **Enter a 5 digit number identifying a customer.**
6. Choose OK, which will display the Map Parameters dialog box.
7. Click OK.

Now that you have defined the parameter, you must include it in the report query before you can use it. You can also display it on your report, since it appears in the Insert Field dialog box.

When the report was initially created, CA-Visual Objects 2.7 constructed an SQL statement based on the servers that you selected in the Report Editor dialog box. The generated statement is as follows:

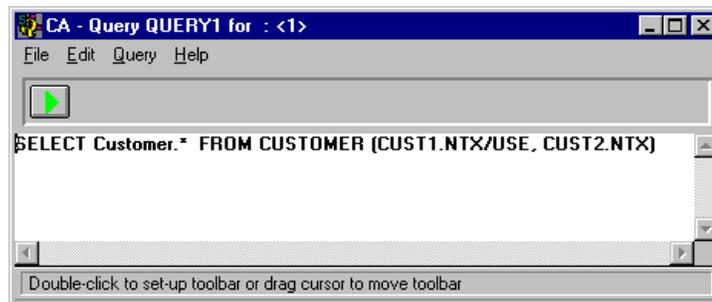
```
SELECT Customer.* FROM ;
       customer.dbf (cust1.ntx/USE, cust2.ntx)
```

In English, this statement reads “include all fields from the customer table and order the records using CUST1.NTX.”

In the steps that follow, you will alter this SQL statement to use the parameter that you defined in the previous steps:

1. Choose Edit Query from the Query menu.

The Edit Query dialog box appears:



2. Move the cursor to the end of the last line.
3. Insert a blank space, and add the following WHERE clause to the existing SQL statement:

```
WHERE ((Cust_ID = '['_parameter1']') OR ('['_parameter1']' IS NULL))
```

Important! Make sure there are no extra spaces between the “[” and “] ” characters.

The WHERE clause serves a dual purpose, allowing the report to be printed either for all customers or a single customer. When a valid customer number is passed (Cust_ID = '['_parameter1']'), the report will include only that customer. If an empty string is passed ('['_parameter1']' IS NULL), the report will include all customers. Although this is a fairly simple query, the Report Editor is also capable of handling very complex queries.

4. Select Close from the File menu to accept the new statement.

The Report Editor will now prompt you for each parameter used in the query.

5. At the prompt, type **00001** and press OK.

6. Select the Print Preview from the File menu. Press OK at the parameter prompt.

The query table will display only the records for customer 00001.

Note: The Report Editor comes with a powerful Query Builder tool. This can help you create queries without knowing SQL.

7. Choose Close to cancel the report.
8. Choose Save from the File menu to save your changes.
9. To close the Report Editor and return to CA-Visual Objects 2.7, double-click the system menu.

Passing Parameters to the Report Editor from CA-Visual Objects

The ReportQueue class, from which the CustList class was derived, allows you to pass parameters to your Report Editor report. To see an example of this, you can add some code designed to work with the Customer List report.

In order for the South Seas Adventure to print the Customer Reports, we will now build it into the printing dialog box:

1. Select the South Seas Adventure Application from the Repository Explorer tree view.
2. Select the New Module toolbar button and enter Tutorial:Reports in the Enter Module Name single line edit.
3. Select the new Tutorial:Reports module.
4. Select the New Entity toolbar button and select Source Code Editor from the local pop-up menu.

The Source Code Editor appears.

5. Add the following Method and Access to the Source Code Editor:

```

METHOD CustomerReport CLASS SSAWindow

LOCAL oDialog AS CustRptDialog
LOCAL oReport AS CustList
LOCAL aParams AS ARRAY

( oDialog := CustRptDialog{ SELF } ):Show()

IF oDialog:lPrintOk
    oReport := CustList{ SELF }
    aParams := { oDialog:cCust_Id }
    // Print only this customer
    DO CASE
    CASE oDialog:nDestination == PRINT_PRINTER
        oReport:Print( aParams )
    CASE oDialog:nDestination == PRINT_SCREEN
        oReport:Preview( aParams )
    CASE oDialog:nDestination == PRINT_FILE
        oReport:SaveToFile(oDialog:ToFileFS:FullPath, ;
            Right(oDialog:ToFileFS:Extension, ;
                3 ), aParams )
    ENDCASE
ENDIF

RETURN SELF

ACCESS CUSTLIST_FILE CLASS CUSTLIST
RETURN "CUSTLIST.RET"

```

This allows the user to include all customers or only a single customer on a report, and also allows the report to print to various destinations.

Note: The dialog box is defined in the Customer:Reports module as `CustRptDialog`. The subclass, `CustRptDialog`, contains additional methods defined to work with this dialog box. You may want to refer to the source code for these methods to completely understand the source code you have just added.

Once the Print Customer Report dialog box closes, an array is created as follows:

```
aParams := {oDialog:cCust_ID}
```

The ReportQueue `Print()`, `Preview()`, or `SaveToFile()` method is then called using this array as a parameter:

```

CASE oDialog:nDestination == PRINT_PRINTER
    oReport:Print(aParams)
CASE oDialog:nDestination == PRINT_SCREEN
    oReport:Preview(aParams)
CASE oDialog:nDestination == PRINT_FILE
    oReport:SaveToFile(..., aParams)

```

The elements of the array will be passed to the Report Editor report—one for each parameter required.

6. When you are finished examining this source code, close the Source Code Editor by double-clicking its system menu and clicking Yes when prompted to save the additions.

Verifying the Results



Verify the results using the following steps:

1. Build the application by clicking the Build toolbar button.
2. Run the South Seas Adventures application by clicking the Execute toolbar button.
3. Log in to the application as usual (Name: **User**, Password: **Trainee**).
4. Select the Customers command from the Reports menu.

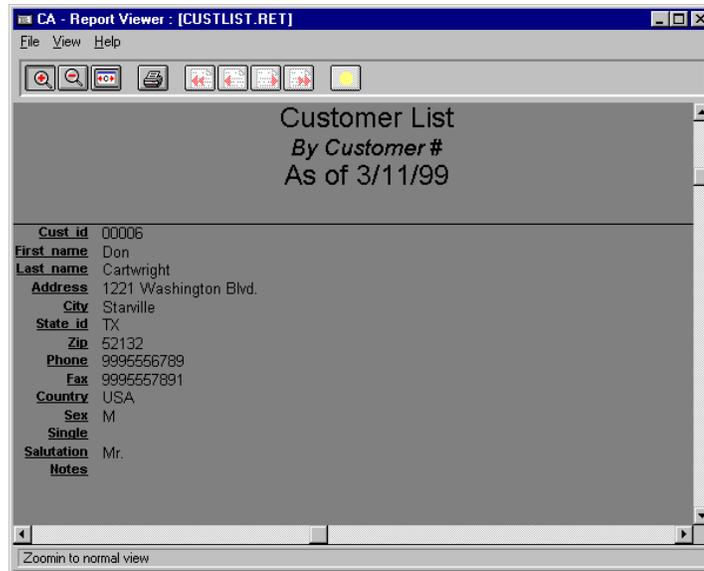
The Print Customer Report dialog box appears.

5. Click the Selection radio button, and choose Cartwright from the Customer # combo box.
6. Click the Screen radio button to print the report to the screen:

A screenshot of the 'Print Customer Report' dialog box. The dialog has a title bar with a close button. It contains two main sections: 'Range' and 'Destination'. In the 'Range' section, there are three radio buttons: 'All', 'Selection', and 'Customer #'. The 'Selection' radio button is selected. Below it is a dropdown menu labeled 'Customer #' with 'Cartwright' selected. In the 'Destination' section, there are three radio buttons: 'Printer', 'Screen', and 'File'. The 'Screen' radio button is selected. To the right of the 'Destination' section, the text 'Default Printer - (.)' is visible. On the right side of the dialog, there are three buttons: 'OK', 'Cancel', and 'Setup'.

7. Choose OK to run the report.

The runtime of the Report Editor is launched, with your report in full view:



8. When you are done, choose the Close command from the File menu to close the Report Editor.

Exit the South Seas Adventures application by double-clicking its system menu, and choosing Yes when prompted.

Summary

In this lesson, you have learned how to use the Report Editor to create and customize a basic report, and how to add parameters to that report. You have also learned how to modify the SQL query (which defines which records will be included in your report), and how to pass parameters to your Report Editor report from your CA-Visual Objects 2.7 application.

In the following lesson, you will learn how to quickly and easily debug your application. You will use the Error Browser to fix compiler errors, and the CA-Visual Objects Debugger to locate and fix logic or runtime errors.

Debugging Your Application

This lesson shows you how to use the Error Browser and Debugger to debug your CA-Visual Objects applications. During this lesson, you will use the Error Browser to quickly fix compiler errors and the CA-Visual Objects Debugger to track down, then fix, logic and runtime errors.

Error Browser Exercise

The Error Browser is the primary tool for locating compiler errors and is automatically displayed after any build if an error occurs during compilation. The Error Browser displays the error information, such as the type and location of the error, and provides easy access to the line in question through the Source Code Editor. By using the Error Browser, you will find that locating and fixing your errors could not be easier.

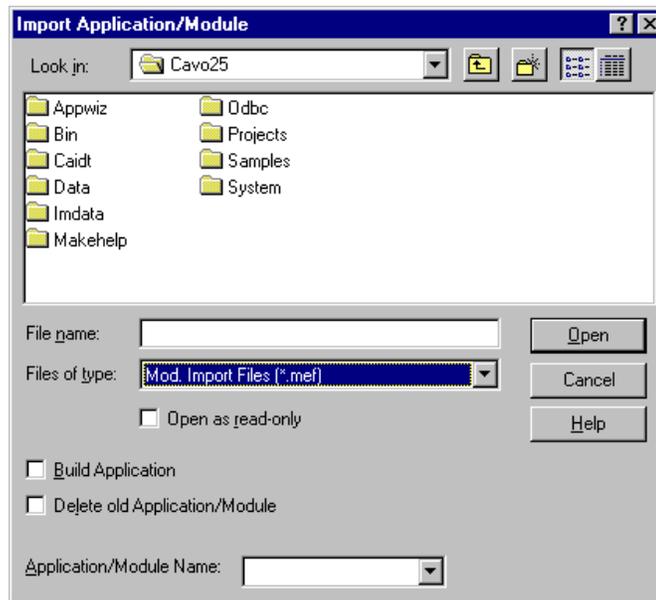
Important! *The errors that occur in this exercise involve reserved words and undeclared variables. Before beginning this exercise, choose the Properties command from the Application menu and make sure the Undeclared Variables check box is not selected on the Compiler tab. (If it is checked, uncheck it.) To close this dialog box, choose OK.*

Importing a Module with Errors

To explore the Error Browser, let's import and compile a module with several intentional errors:

1. Open the South Seas Adventures application by double-clicking its branch in the Repository Explorer tree view.
2. Select the Import command from the File menu.

Select (*.mef) from the **Files of Type** drop-down list box:



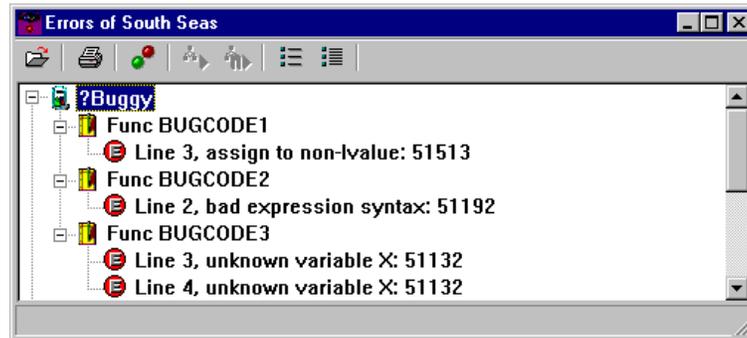
4. Select the file `_BUGGY.MEF`, which is in your CA-Visual Objects 2.7 `SAMPLES\SSATUTOR\FILES` subdirectory, then choose Open.

The ?Buggy module button appears as the first module in the SSA application tree.



- Choose the Build toolbar button to build the application.

Since there are errors in the code, you are presented with the Error Browser. The compiler errors are displayed in a tree-like structure that can be expanded or collapsed:



It is similar to the application's tree view in the Repository Explorer. In this structure, the first level indicates the module, the second level indicates the entity, and the third (and final) level indicates the line number and error message.

Note: The question mark in the module name was replaced with an underscore when it was exported as an .MEF file, to conform to the file name convention rules.

Resolving the Errors

Now, use the Error Browser to resolve the errors in this code.

- Double-click the first error, which appears as follows:

 **Line 3, assign to non-lvalue: 51513**

This brings you to the Source Code Editor with the cursor located on the line where the error occurred:

```
LOCAL Date := Today() AS DATE
```

The error occurs because DATE is a reserved word and, therefore, cannot be used as a variable.

- Fix this code by changing the variable Date to **dDate** in the last two lines of code, as follows:

```
LOCAL dDate := Today() AS DATE
RETURN (dDate)
```

- Close the Source Code Editor by double-clicking its system menu and answering Yes when prompted to save your changes.

You are returned to the Error Browser and, even though you have resolved the first error, it still appears in the error browser.

Double-click the next error:

E Line 2, bad expression syntax: 51192

Again, you are brought directly into the Source Code Editor with the cursor located on the line where the error occurs:

```
nValue := nValue +
```

The error message indicates that this line contains bad expression syntax and it indeed has an incomplete statement.

Correct the statement by adding **10** to the end of the line, as follows:

```
nValue := nValue +10
```

- Close the Source Code Editor by double-clicking its system menu and answering Yes when prompted to save your changes.

Once again, you are returned to the Error Browser.

The next two errors are referring to the same problem:

E Line 3, unknown variable X: 51132

E Line 4, unknown variable X: 51132

This error message indicates that the variable *X* is unknown to the compiler, which means that it is not declared.

Double-click the first of these two errors.

- Remove the comment indicators (*//*) from the LOCAL statement, just above the line you are currently on, to correct the problem:

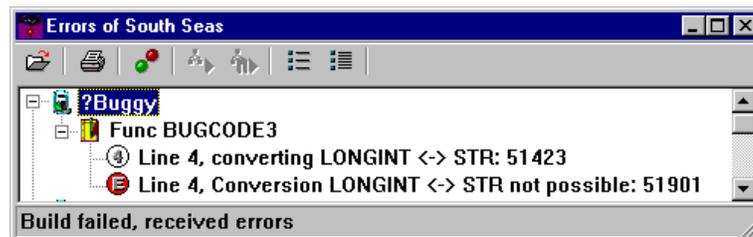
```
LOCAL X AS INT
```

- Close the Source Code Editor by double-clicking its system menu and answering Yes when prompted to save your changes.

- Choose the Build toolbar button to rebuild the application.



The Error Browser now appears with another error:



The previous correction causes the compiler to discover a new error. This error involves a type conversion that is not possible.

11. Double-click the error to see the line in question.

By examining this line you can see that the function, `FUNCTION BugCode3()`, returns an `INT`, although it was declared to return type `STRING`. This is a good example of why strong typing is beneficial at the development stage—an error of this nature, although very obvious at compile time, might easily elude you at runtime.

12. Fix the error by converting the return value of the function to a string, as follows:

```
RETURN Str(X)
```

13. Close the Source Code Editor by double-clicking its system menu and answering Yes when prompted to save your changes.



14. Choose the Build toolbar button to rebuild the application, which should now compile without errors.

Note: It is possible that you may still receive warnings, such as, “Line 1, Variable UVALUE is declared but never used: 51455.” These particular warnings can be ignored, as they will not adversely affect the application.

15. Close the Error Browser by double-clicking its system menu and return to the Repository Explorer.

Debugger Exercise

The CA-Visual Objects Debugger is the primary work space in the IDE for tracking and correcting errors that occur at runtime. To access the Debugger from the various browsers and editors, use either the Debug Run menu command or the Trace Expression toolbar button.

By using the Debugger you can:

- Control the execution of your application while viewing the source code in the Debug Source Code window
- Execute any part of your application using one of several execution modes, including a mode in which you step through the code one line at a time
- Conditionally stop program execution using breakpoints
- Monitor watch expressions in a separate window
- Evaluate expressions on-the-fly
- View and modify variables of all storage classes
- View database, index, and other work area information in a separate window
- View and modify system settings

The most common type of errors that you will correct using the Debugger are known as *logic errors*. Logic errors occur when the application does not perform as intended, but does not crash or give any other immediate indication of flaw (such as a runtime error message). These types of errors are often the hardest to locate and an area in which the Debugger excels.

Viewing the Error

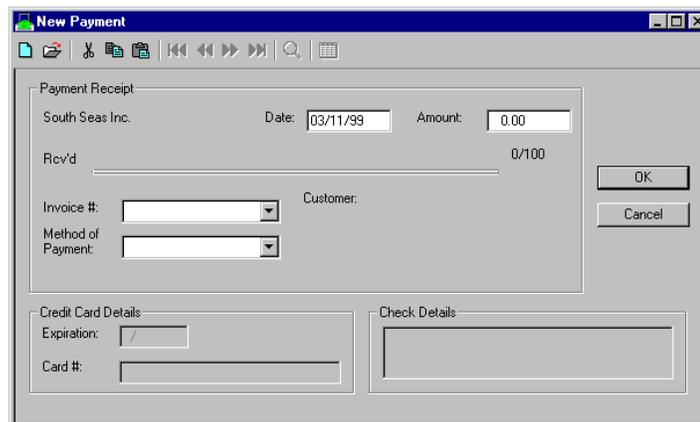
In this exercise, use the Debugger to track down an error that will be added to the South Seas Adventures application:



1. Provided the application was successfully built at the end of the previous exercise, you can run it now by choosing the Execute toolbar button.
2. Log in to the application as usual (Name: **User**, Password: **Trainee**).
3. Choose the New command from the File menu.
4. Select the Payment radio button from the New Record dialog box, as follows:



5. Choose OK to display the New Payment window:



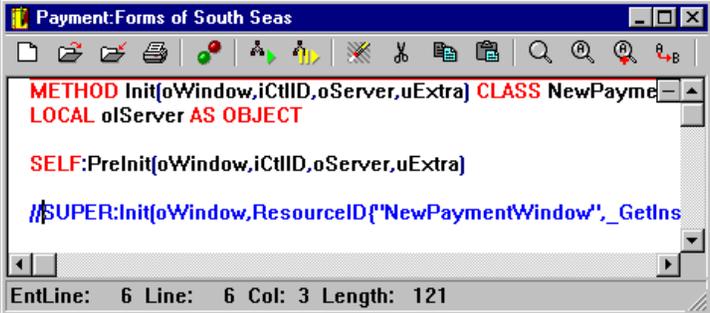
As you can see, the window starts as expected. We will now proceed to break this so that we can debug the problem.

6. Close South Seas Adventures by double-clicking its system menu and choosing Yes when prompted.

Introducing an Error

To introduce the error:

1. Open the Payment:Forms module by clicking on it in the Repository Explorer tree view.
2. Open the NewPaymentWindow:Init method by double-clicking on its icon.
3. Place two comment indicators (//) at the beginning of line 6 so that the SUPER:Init method becomes a comment line:



```

METHOD Init(oWindow,iCtlID,oServer,uExtra) CLASS NewPayme
LOCAL olServer AS OBJECT

SELF:PreInit(oWindow,iCtlID,oServer,uExtra)

//SUPER:Init(oWindow,ResourceID('NewPaymentWindow',_GetIns

```

EntLine: 6 Line: 6 Col: 3 Length: 121

4. Close the Source Code Editor by double-clicking its system menu and answering Yes when prompted to save your changes.
5. Choose the Build toolbar button to rebuild the application.

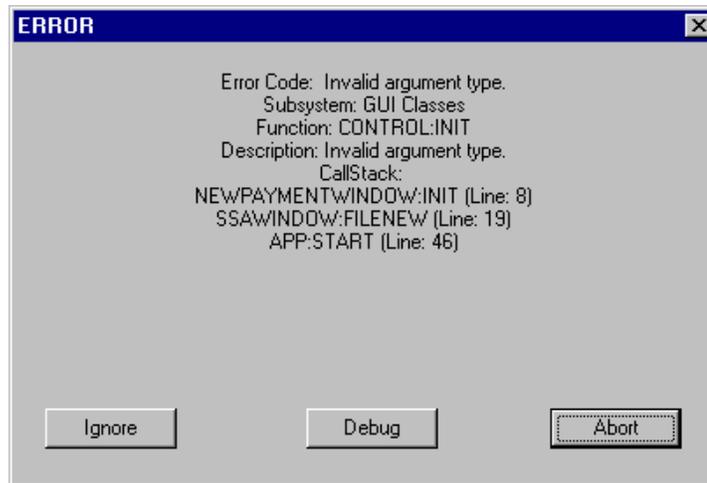


Now it is time to see the results of our actions:

1. Provided the application was successfully built at the end of the previous exercise, you can run it now by choosing the Execute toolbar button.
2. Log in to the application as usual (Name: **User**, Password: **Trainee**).
3. Choose the New command from the File menu.
4. Select the Payment radio button from the New Record dialog box.



- Choose OK to try to display the New Payment window and you should have an error that looks like this:



Although this does not look very helpful it is giving us more information than you would think:

- There is an invalid argument—one of the parameters being passed to something else.
- It is invalid when it is passed to the CONTROL.INIT method. Therefore, it is in a control.
- The error has happened by the time it gets to line 8 of the NewPaymentWindow:Init method—this is a good place to start.

Press the Abort button to clear this window and press the OK button on the next window to acknowledge the end of execution.

Set Debugging On

The CA-Visual Objects Debugger can be set at three levels:

Application level—provides debugging information for the entire application.

Module level—provides debugging information for every entity in the current module.

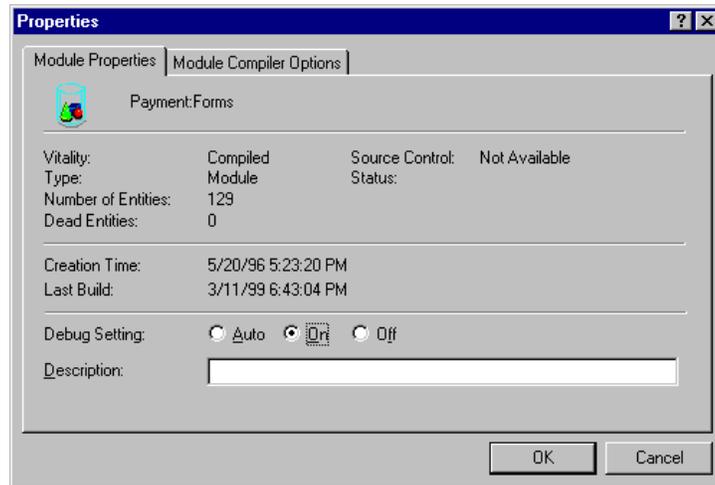
Entity level—provides debugging information only for the current entity.

Set Debugging at the Module Level

The South Seas Adventures application currently does not have debugging activated; therefore, you need to attach debugging information to the entities or modules that you feel are affected. To save you some time searching through code we will follow the clues from the error message that we received.

We know that it happens in the `NewPaymentWindow:Init` method so the easiest way to deal with this is to attach debugging to the module as follows:

1. From the Repository Explorer, select the `Payment:Forms` module and click the right mouse button.
2. Select the Properties command from the local pop-up menu and select the **On Debug** radio button on the Properties dialog box:



3. Press the OK button.

Notice that the Debug column of the Repository Explorer Details List View shows `D<On>`, telling you that debugging is turned on for this module, and that all of the methods in the module are now marked as uncompiled.



4. Choose the Build toolbar button to rebuild the application.

Running the Application Using the Debugger

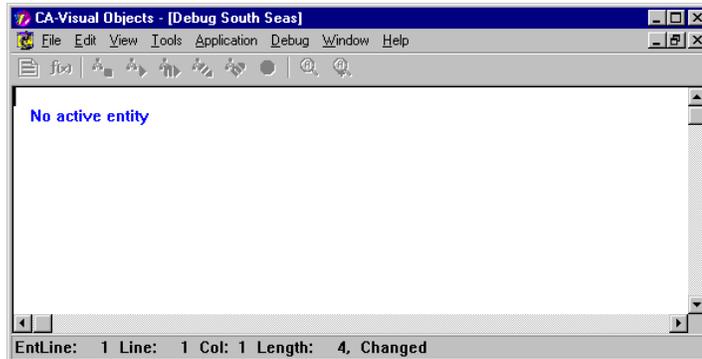
Now you are ready to debug the application. First, clear some space on your Windows desktop so that you can see the Debugger while the application is running:

1. Size and position the CA-Visual Objects window so that it takes up only the top half of your screen.



2. Select the Debug toolbar button.

CA-Visual Objects will now start the debug window:



The South Seas Adventures splash screen is displayed:



No code will be displayed in the debug window until it reaches the first line of code with debug turned on.

3. Log in to the application as usual (Name: **User**, Password: **Trainee**).
4. Size the South Seas Adventures application window, so that it takes up the bottom half of your screen and CA-Visual Objects is visible in the top half of your screen.

Locating the Bug

You are now ready to debug the application.

1. From the South Seas Adventures main window, select the New command from the File menu.

2. Select the Payment radio button from the New Record dialog box:



3. Press the OK button to open the Payment window.

Before the window can open you should see the CA-Visual Objects window change and the debugger start to operate. The application is now at a standstill waiting for some instruction from the debugger.



Some points to note about the debugger:

1. The solid black line with white text is the line that will be executed next, which, in this case, is the first line of the Init method.
2. When the debugger is expecting to be in control, the toolbar buttons are in color. If any action is taken in the application at this time, the debugger will lose its place and the results of any tests will not be correct.

Evaluating Expressions

Now you can inspect the variable values:



1. Choose the Evaluate toolbar button.
The Evaluate Expression dialog box appears.
2. Type **SELF** in the Expression edit control. This represents the class NewPaymentWindow.
3. Choose Evaluate.
4. The Result edit control displays something similar to:
{{(0x020C)0x032B4C90} CLASS NEWPAYMENTWINDOW}
This shows that the class has already been created.
5. Choose Cancel to close the window and return to the CA-Visual Objects window.

Let's look at all the local variables now:

1. Select the View Locals command from the Debug menu.

The Local/Private Variables window displays. Expand the SELF variable by clicking on the file icon. Scroll down through the list of variables and you will see that they are all <No Object> or NIL, which means that they have not been initialized yet. The initialization would normally take place during the SUPER:Init which we have commented out.

2. Close the Local/Private Variables window.



3. Choose the Step In toolbar button to enter into the entity on the line that is currently executing.

This will move you to the PreInit line.



4. Click the Execute Next Line toolbar button (or select the Step command from the Debug menu).

This will execute the PreInit line and move us to the next line of code to execute, having bypassed the SUPER:Init line that we commented out.

We are now on line 8, which we know is the line where the error will occur; and we know that it is a problem with one of the parameters.

Let's look at all the local variables now:

1. Select the View Locals command from the Debug menu.

The Local/Private Variables window displays. Expand the SELF variable by clicking on the file icon. Scroll down through the list of variables and you will see that they are still set to <No Object> or NIL, which is not surprising.

Close the Local/Private Variables window.



2. Click the Execute Next Line toolbar button (or select the Step command from the Debug menu).

This will display the error as before and we can now understand that passing SELF as the first parameter is not going to work. The init of the control will fail because its owner, NewPaymentWindow, has not been initialized.

3. Press the Abort button to clear this window and press the OK button on the next window to acknowledge the end of execution. This will end the debug session and return you to the Repository Explorer.

From what we have just looked at it should be evident that the Init method for the super class has not been called.

Correcting the Error

Now that you have found the problem, you can return to the Source Code Editor for the NewPaymentWindow:Init method. Remove the comment lines that you inserted earlier save the code and close the editor.



Rebuild the application using the Build toolbar button.

Summary

This lesson provided you with an understanding of some of the key features of the error detection and debugging facilities provided in CA-Visual Objects. You used the Error Browser to diagnose and fix compiler errors, and the Debugger to resolve a runtime error.

Adding Help to Your Applications

This lesson shows you how to implement an online help system for your CA-Visual Objects 2.7 applications.

Overview

An online help system can mean the difference between the success and failure of your application. Applications are often judged on the existence and quality of their online help.

The creation of an online help system is non-trivial. Its design should be included at the start of your application's development.

When designing an online help system, you must decide upon the level of detail you want to provide and how best to organize the hierarchy of the help topics. Once the system is designed, you must provide the written text and build the help system.

Using the Windows help system, installed with every copy of Windows, offers several advantages to you. First of all, it automatically loads the help system within your application when context-sensitive help is requested. Additionally, users can be expected to have some familiarity with its operation because most Windows applications provide an online help system.

Context-Sensitive Help

CA-Visual Objects 2.7 not only provides the capability for attaching help to your application, but also provides the mechanism to invoke *context-sensitive* help. Context-sensitive help is based on the current state of the application, or *context*, in which the user requests help.

F1

Pressing the F1 key invokes help based on the menu command or control that currently has focus. For example, to obtain help on a menu command or control, the user need only highlight it and press F1.

If no control or menu command has focus when F1 is pressed, the default behavior is to display the Contents topic defined in the associated help file.

Shift+F1

Pressing Shift+F1 allows the user to select an item using a special help cursor that consists of an arrow and a question mark. To get help on a control or menu command, the user presses Shift+F1, then locates and clicks the control or menu command with the help cursor.

If no help is available for the item selected, the default behavior is to display the Contents topic defined in the associated help file.

Exercise

During this exercise, you will discover how the South Seas Adventures help system is implemented. You will also learn where in your application to specify the associated help file, and how to set context-sensitive help—for a window, a control in a window, and a menu command.

Implementing Context-Sensitive Help

Typically, an application has only one help file for the entire application. When you assign a help file to a shell window, each of its child windows, unless another help file is specified, uses the same help file as its parent. However, the architecture used by CA-Visual Objects 2.7 makes it very easy to define a separate help file for each window.

Attaching Your Help File

Let's see how attaching a help file is accomplished in the South Seas Adventures application:

1. Double-click the South Seas Adventures application branch on the Repository Explorer.
2. Open the SSA Shell:Forms module by clicking its branch in the Repository Explorer tree view.
3. Find the SSAWindow window entity and double-click it to open the Window Editor.

4. Find the Help File Name property in the Shell Window Properties window (under the ShellWindow tab):



5. The Help File Name property already has **SSA.HLP** assigned to it. This generated the following line of code in the Init() method of the SSAWindow:

```
SELF:HelpDisplay := HelpDisplay{"SSA.HLP"}
```

The HelpDisplay class in this code establishes a link between the South Seas Adventures shell window (SSAWindow) and the SSA.HLP help file.

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

HelpRequest Event System

When help is requested by the user, a HelpRequest event is generated. The CA-Visual Objects dispatcher then invokes the window's HelpRequest() method, passing it a HelpRequestEvent object. The HelpRequestEvent class is used to describe the context, and item combinations, for which help is requested.

The default HelpRequest() method (from the Window class) invokes the help file, based on the information in the HelpRequestEvent object:

```
METHOD HelpRequest(oHRE) CLASS Window
...
SELF:HelpDisplay:Show(oHRE:HelpContext)
...
RETURN NIL
```

The HelpDisplay:Show() method invokes WinHelp, with the *oHRE:HelpContext* parameter representing the keyword that WinHelp searches for in the help file.

Help Context Property

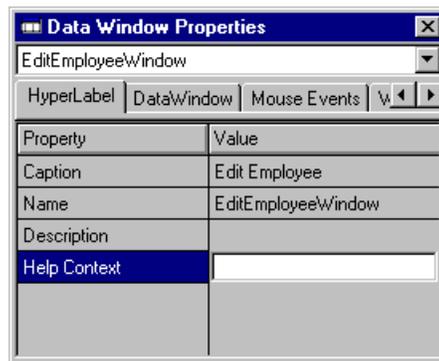
You can specify Help Context strings for windows, menu commands, controls, and field specs. The Help Context property is the means by which CA-Visual Objects enables your application to supply context-sensitive help. CA-Visual Objects extracts the Help Context property of the item (such as control or menu command) for which the user requests help. This Help Context is then used in the call to WinHelp.

WinHelp performs a keyword search for a topic that matches the Help Context string. If a match is found, the topic displays. If you do not specify a Help Context property for the specified item, or if the Help Context specified is not found, WinHelp displays the Contents topic of your help file.

Assigning Help to a Window

The South Seas Adventures help file has a topic, called Employees, which describes how to edit employee data. If the user requests help when an Edit Employee window is open, you want WinHelp to display the Employees topic. The following steps show how this is accomplished:

1. Open the Employee:Forms module by clicking its branch on the Repository Explorer.
2. Locate and double-click the EditEmployeeWindow entity to open the Window Editor.
3. Scroll through the Data Window Properties window until you see the Help Context property (under the HyperLabel tab):



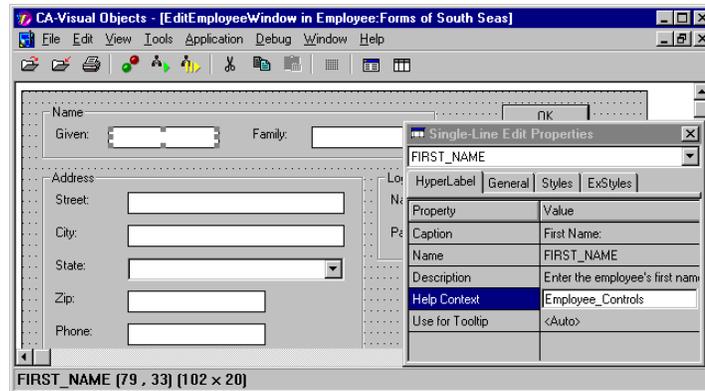
4. Set the Help Context property to Employees.

When help for this window is requested, WinHelp now displays the Employees topic.

Assigning Help to a Control

The South Seas Adventures help file has another topic, called Employee Controls, which describes all of the controls in the EditEmployee window. When help is requested for any of these controls, the following steps enable WinHelp to display this topic:

1. Select the GIVEN edit control in the upper-left corner of the EditEmployee window:



2. Select the Help Context property from the Single-Line Edit Properties window (under the HyperLabel tab). Replace the underscore in the current string (Employee_Controls) with a space to make it resemble other single-line edit controls.
3. The Help Contexts for the other fields have already been supplied with the same value, so choose the Save toolbar button to save your changes.
4. Close the Window Editor by double-clicking its system menu.



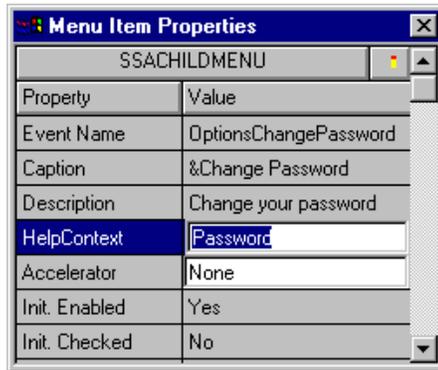
Assigning Help to a Menu Command

The South Seas Adventures help file has another topic, called Password, which describes how to change the current user's password. The following steps show how this help topic is associated with the Change Password command in the Options menu:

1. Open the SSA Child:Menu module by double-clicking its branch on the Repository Explorer.
2. Find the SSACHildMenu menu entity and double-click it to open the Menu Editor.

3. Scroll through the Menu Editor and click the Change Password menu item under the Options menu.

The Help Context property for this item is already set to Password, as shown in the Menu Item Properties window:



4. Close the Menu Editor by double-clicking its system menu.

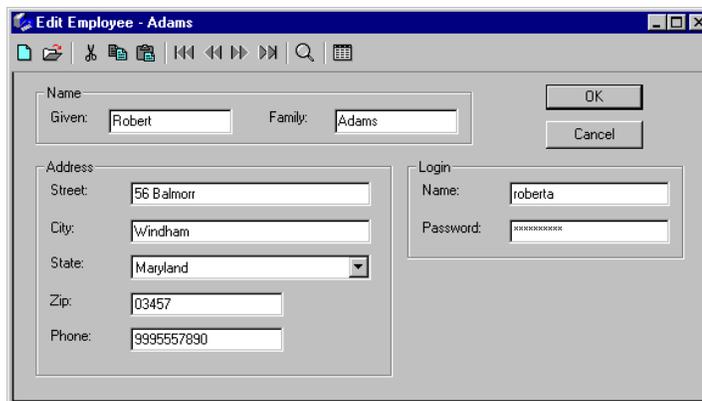
Invoking Context-Sensitive Help

Now, let's test the context-sensitive help for your application. First, open the Employee Browser:



1. Choose the Build toolbar button.
2. Choose the Execute toolbar button.
3. Log into the application as usual (Name: **User**, Password: **Trainee**).
4. Select the Open command from the File menu.
5. Click the Employee radio button, and choose OK.
6. From the Employee Browser, choose Edit.

The Edit Employee window displays:



Viewing Help for a Control

This section shows you how to obtain help for any edit control in the employee window:



1. Press Shift+F1 to display the help cursor.
2. Click in any of the edit controls.

The South Seas Adventures Help window appears, displaying the Edit Employee Controls topic:



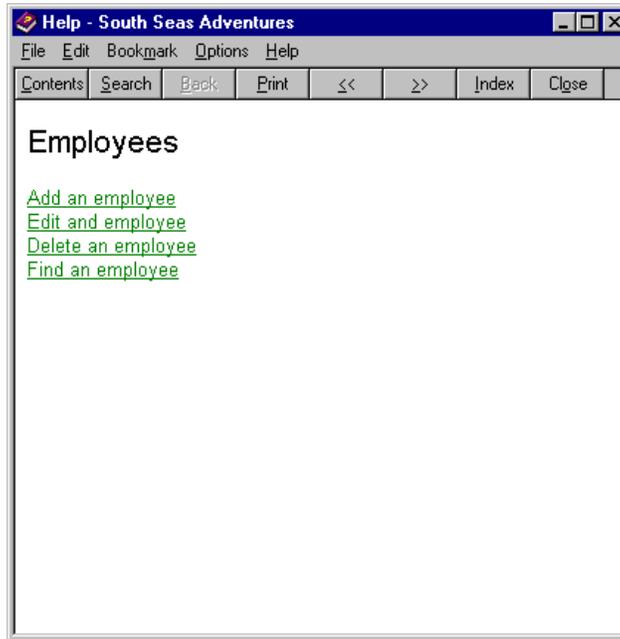
3. When you are finished reviewing the help text, close the Help window by double-clicking its system menu.

Viewing Help for a Window

This section shows you how to obtain help for the entire Edit Employee window:

1. Press Shift+F1 to display the help cursor.
2. Click the Edit Employee window's title bar.

The South Seas Adventures Help window appears, displaying the Employees topic:



3. When you are finished reviewing the help text, close the Help window by double-clicking its system menu.
4. Choose Cancel to close the Edit Employee window.
5. Choose Close to close the Employee Browser.

Viewing Help for a Menu Command

To view help for a menu command:

1. Select the Options menu and position the mouse pointer over the Change Password option.
2. Press F1 to call the help system.

The South Seas Adventures Help window appears, displaying the Change Password topic.

3. When you are finished reviewing the help text, close the Help window by double-clicking its system menu.
4. Close the application by double-clicking its system menu and answering Yes when prompted.

Implementing Direct Calls to Help

Many Windows applications provide their users with help from other sources. Some of these sources are menu commands, button controls, and event processes.

Menu Commands

Help can be invoked in response to a menu command selection. To implement help from a menu command, you can:

- Trap menu selection events using the `MenuSelect()` event handler method
- Create a separate menu event method for the menu command

This section shows how the South Seas Adventures application creates a separate menu event method for a menu command:

1. Choose the SSA Shell:Forms module from the Repository Explorer.
2. Find the `SSAWindow:HelpContents()` method in the list view and double-click it to open the Source Code Editor. The following code displays:

```
METHOD HelpContents() CLASS SSAWindow
    SELF:HelpDisplay:Show("HelpIndex")
RETURN SELF
```

This method illustrates how to call the help system directly using the `HelpDisplay:Show()` method. You can pass a help context keyword as the parameter for `HelpDisplay:Show()` function or, as shown in this example, one of the following reserved keywords to access a standard help feature:

Help Context	Help
HelpIndex	Displays the Contents topic as defined in the application help file
HelpOnHelp	Displays the Contents topic as defined in your Windows help file (normally WINHELP.HLP, in which the Contents topic is How to Use Help)

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

Creating Help Files

Creating Windows help support for your application is a three-step process:

1. Create the required help system source files, such as topic and project files.
2. Compile the source files using a help compiler. This creates a help file that is ready to be used by the Windows Help.
3. Create the links in your application that use the help system.

Topic Files

A topic file contains the text of your help file for one or more topics. The topic file also contains the codes needed to link topics together.

The help compiler requires topic files coded in Rich Text Format (RTF). To create a file in this format, you can use a simple ASCII text editor, or you can use a word processor that can export files in this format.

Using a simple text editor requires you to explicitly code your help topics using the rich text coding syntax. On the other hand, using a word processor that can export in RTF format can reduce your task dramatically. Using simple formatting commands and footnotes to create the document, you then export it in RTF format. The document is translated into the appropriate commands and codes that can then be compiled by the Help compiler.

The South Seas Adventures topic file, SSA.DOC, was created using Microsoft Word for Windows. The file was then exported to rich text format as SSA.RTF. You can retrieve the SSA.RTF file with your program editor to view the RTF codes. These files can be found in your CA-Visual Objects 2.7 SAMPLES\SSATutor\Help subdirectory.

Project File

The project file contains a list of source files required to create the help file. It also contains window definitions and compiler directives.

The South Seas Adventures help project is stored in the SSA.HPJ file. (Also in our CA-Visual Objects 2.7 SAMPLES\SSATutor\Help subdirectory.)

Summary

In this lesson, you have implemented online help for a CA-Visual Objects application and, through the Help Context property, you now know how to implement context-sensitive help. You have also seen how to make calls directly to the Windows Help system.

Using Win32 API Functions

In this lesson, you will learn how to call Win32 API functions directly from CA-Visual Objects 2.7. This will demonstrate how to expand the functionality of your applications, as well as illustrate how Windows programs work on a low level.

Overview

What Is the Win32 API?

Windows provides more than just the graphical shell in which our applications run. It also consists of over 600 built-in functions available to any Windows program at runtime. These functions range from creating and manipulating windows and menus, to simple network functions. The Win32 API also provides predefined data structures and message definitions that your programs can use.

CA-Visual Objects provides the capability to use as little or as much of the Win32 API as we wish. Indeed, we can write entire CA-Visual Objects programs using only Win32 API calls. However, we would lose the ability to use many of the object classes provided. If a function is provided by both CA-Visual Objects and the Win32 API, it is preferable to use the CA-Visual Objects function.

Calling Win32 API Functions

Calling Win32 API functions is as simple as calling any other function. The only stipulation is that the function's prototype must be provided. The Win32 API library does precisely this, as well as defining Windows data structures and messages. For example, this is the prototype for the `MessageBox()` function, which provides a simple modal dialog box:

```
_DLL_FUNC MessageBox( ;  
    dwHandle As DWORD, ;  
    cMsg AS STRING, ;  
    cTitle AS STRING, ;  
    dwFlag AS DWORD) ;  
AS INT PASCAL :USER32.MessageBoxA
```

The `_DLL_FUNC` statement indicates that the `MessageBox()` function resides in an external DLL. The parameters for `MessageBox()` are the parent window's handle (a unique identifier within Windows), the message text and caption as strings, and the flags indicating the type of message box desired. The function returns a value which is defined as a long integer, in this case a number corresponding to the user's selection from the message box. The function follows the Pascal calling convention and resides in the `USER32` library.

Calling this function is as simple as:

```
MessageBox(SELF:Handle(), ;  
  "This is a test message.", ;  
  "Test", _OR( MB_ICONINFORMATION, MB_OK))
```

Fortunately for us, CA-Visual Objects has its own classes built around `MessageBox()` so that we can call different types of boxes. See `ErrorBox`, `WarningBox`, `InfoBox` and `TextBox` in the help file.

Exercise

Let's begin this lesson by examining one of the Win32 API functions:

Function Name	Purpose
<code>GetSystemMetrics()</code>	Retrieves various system settings

Windows Metric Information

The `GetSystemMetrics()` function retrieves information about various system metrics (for example, the heights and widths of various elements displayed by Windows). You will now examine the use of this function to set the size of a window to the full size of the screen without maximizing the window:

1. Open the SSA Application by double-clicking its branch in the Repository Explorer.
2. Open the SSA Shell:Forms module clicking its branch.
3. Open the SSAWindow:PostInit() method by double-clicking its entity in the list view. Note the function call assigning the return value from the function `FullWinSize()` to the window's size attribute. This code makes the window occupy the entire screen without having to be maximized:

```
SELF:Size := FullWinSize()
```

FullWinSize(), defined in the App:Misc module, determines the actual maximum size of the window by calling the Windows function **GetSystemMetrics()**:

```
FUNCTION FullWinSize() AS Dimension
    LOCAL oDim AS Dimension, ;
           nHeight AS SHORTINT, ;
           nWidth AS SHORTINT

    nHeight := GetSystemMetrics(SM_CYSCREEN)
    nWidth  := GetSystemMetrics(SM_CXSCREEN)

    oDim := Dimension{nWidth, nHeight}

    RETURN oDim
```

The parameter passed to **GetSystemMetrics()** indicates which system value is to be returned. In this case **SM_CXSCREEN** instructs **GetSystemMetrics()** to return the width of the screen, and **SM_CYSCREEN** returns the height of the screen. In **FullWinSize()**, these two values are combined as a **Dimension** object, which is used to set the size of a window.

4. Close all open Source Code Editors by double-clicking the appropriate system menus.

Summary

In this chapter, you have learned about the Win32 API functions and how to use direct calls to the Win32 API in your CA-Visual Objects applications.

In the next lesson, you will be able to create a library and a DLL in order to share code among your applications.

Using Libraries and Dynamic Link Libraries

In this lesson, you will be introduced to the benefits of sharing code among your applications using libraries and dynamic link libraries (DLLs). Not only will you learn how to create a library and a DLL, as well as the circumstances in which to use each, but you will be able to produce a .DLL file that can be distributed easily to other programmers or to your end users.

Overview

Both libraries and DLLs provide a means of sharing code among your applications. We explore each type of library in the following sections.

Libraries

You can use a library (or *shared* library) in the same way a .LIB file is used in developing character mode applications. Many of your CA-Visual Objects applications may include the same library and share copies of its code. This library is then linked into the executable (.EXE) file when your application is generated.

Although a library is created much like an application, you cannot run a library as a stand-alone application or create an executable (.EXE) file from one.

The source code of libraries created in CA-Visual Objects may be distributed by exporting the source code to an Application Export Format (.AEF) file; however, this is not usually desirable. Another option is to create a DLL, which provides more flexibility and several advantages.

Dynamic Link Libraries

A DLL is maintained in an external file distinct from any other application. It contains compiled and linked code that can be called from and shared by many .EXE files. DLLs also allow your applications to make better use of available memory, because they are loaded only once, even when shared by many applications.

Memory Efficiency	Using DLLs, you give your applications more memory for data. An .EXE file owns, and can access, a data area called DGROUP (also referred to as <i>near data</i> or <i>static data</i>) that typically holds your STATIC variables and character constants. This area is unlimited in size.
Library Distribution	DLLs are an ideal means for distributing your work to other developers without giving away your source code. When you create a .DLL file from the source code in your repository, a corresponding .AEF file is also generated, defining the <i>public protocol</i> for the DLL. The only source code you distribute is the .AEF file, which contains only _DLL declaration statements that point to entities in the .DLL file by name. Once imported, this .AEF file is created as a library (rather than a DLL) that can be included in the search path of any application needing access to the .DLL file.
Ease of Application Maintenance	DLLs make it easier to maintain your code. Since DLLs are linked only at runtime, you can replace a .DLL file without suffering the consequences of having to relink your .EXE file (provided that the public protocol of the DLL has not changed). You will also save development time, since you do not have to rebuild the entities contained in a DLL whenever you rebuild an application.

Exercise

This exercise demonstrates how to create a library and a DLL from existing code in the South Seas Adventures application. In working through the steps, you will also learn how to generate a .DLL file and use the corresponding .AEF file defining its public protocol as a library.

Creating and Using a Library

A library is created in much the same way as an application. It does not, however, contain a Start() entity since the code of a library is not intended to be run as a stand-alone application.

In this exercise, you are going to create a library called MyLib. You will transfer the IniFile module of the South Seas Adventures application into the library, which can then be used in other applications.

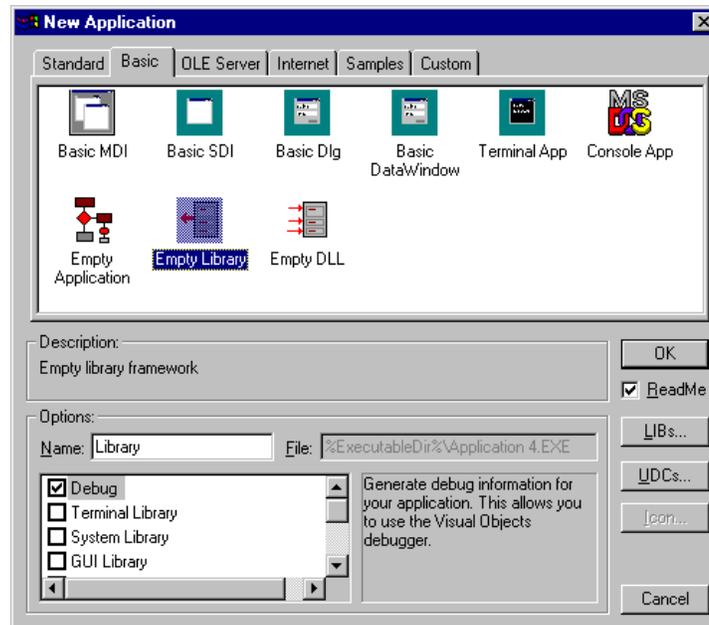
Creating a New Library Application

To create a new library application:

1. From the Repository Explorer, select the Default Project.
2. From the Repository Explorer, choose the New Application toolbar button.

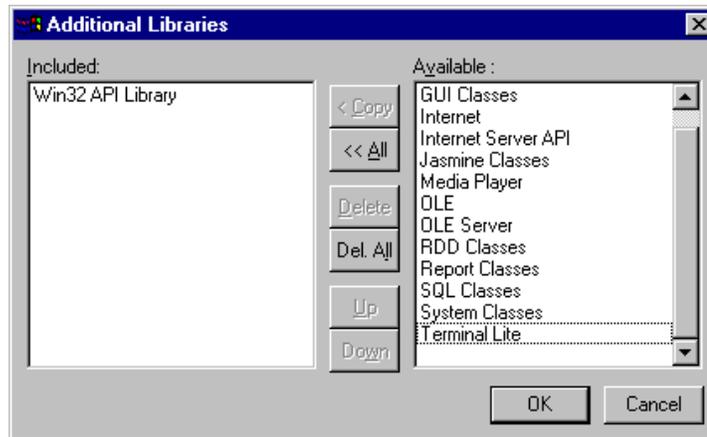


The Application Gallery dialog box appears:



3. Click on the Basic tab and select the Empty Library icon.
4. Type **MyLib** in the Application Name edit control.
5. Check the boxes to include the RDD Library and the System Library.

- Since some of the functions in the new library will make Win32 API calls, include the Win32 API library in the search path by pressing the LIBs push button, scrolling through the Available list box and double-clicking on Win32 API Library:



- Choose OK to close this dialog window.
Note that there is no path specified for a library application.
- Choose OK.
The empty library displays in the tree view.

Moving Modules Between Applications

You now need to return to the South Seas Adventures application to move the IniFile module to MyLib in the tree:

- Open the South Seas Adventures application by double-clicking its branch on the Repository Explorer tree view.
- Scroll through the modules and select the IniFile module by clicking its branch.
- Click and hold down the left mouse button on the IniFile module and drag the module to the MyLib library, then release the mouse button.

The IniFile module is moved to the MyLib library.

- Depending on your default system settings, an empty default module, Module1, may exist in the MyLib library. If so, select the Module1 module from MyLib and press the delete key. When prompted select **Yes** to continue.

Notice that all of the modules in the South Seas Adventures application that were using any of the functions in the IniFile module need to be rebuilt. This is because they have lost their link to the functions you moved to the library. You can fix this by including MyLib in the application's search path and rebuilding the library, as shown next.

Building the Library

In order to use the library, you must first build it:

1. Select the MyLib Library by clicking it.
2. Choose the Build toolbar button.



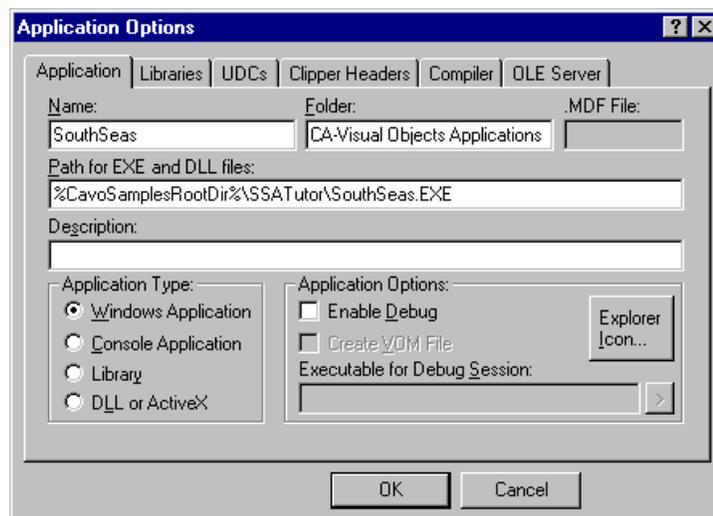
Using the Library

To use a library in an application, you must include it in the application's search path. Let's try this using your library for the South Seas Adventures application:

1. Select the South Seas Adventures module by clicking its branch.
2. Choose the Application Properties toolbar button.



The Properties dialog box appears:



3. Select the Libraries tab.
4. Go through the Available list box and select MyLib by double-clicking it. MyLib will move from the Available list box to the Included list box.
5. Choose OK.
6. Rebuild the South Seas Adventures application by clicking the Build toolbar button again.



The South Seas Adventures application is now ready to be executed and will run just as before. Where it used to access code from its own IniFile module, it now accesses code from the library, MyLib.

Creating and Using a DLL

If many of your applications use common code, it would be wise to provide this code as one or more DLLs. At runtime, a .DLL file is shared by the applications that use its code.

This exercise will take you through the steps to create a DLL from a library and use it from the repository. It will also show you how to create a stand-alone .DLL file which can be easily distributed and maintained. Lastly, you will learn how to create a foreign-hosted DLL to be used with non-CA-Visual Objects applications.

Creating a New DLL Application

You are now going to create a DLL from the MyLib library, as follows:



1. From the Repository Explorer, select the Default Project by clicking its branch.
2. From the Repository Explorer, choose the New Application toolbar button.
3. Click on the Basic tab and select the Empty DLL icon.
4. Type **MyDLL** in the Application Name edit control.
5. Check the boxes to include the RDD Library and the System Library.
6. Since some of the functions in the new library will make Win32 API calls, include the Win32 API library in the search path by pressing the LIBs push button, scrolling through the Available list box and double-clicking on the Win32 API library. Then, click OK.
7. Type in the path and the file name for the DLL (for example, %CavoSamplesRootDir%\SSATutor\MyDLL.DLL).
Choose OK.

Copying Modules Between Applications

You now need to return to the MyLib library to copy the IniFile module to MyDLL in the tree:

1. Open the MyLib library by double-clicking its branch in the Repository Explorer Tree View.
2. Select the IniFile module by clicking its branch.
3. Right click and select Edit All Source In Module.
4. Choose Select All from the Edit menu.
5. Choose Copy from the Edit menu.

6. Close the Source Code Editor.
7. Select the Module1 module in the MyDLL library.
Note: If there is no Module1 in MyDLL, select New Module from the File menu, then click OK to create Module1.
8. Open the Source Code Editor and select Paste from the Edit menu.
9. Close the Source Code Editor by double-clicking on the System icon and answering Yes when prompted to save.
10. Select the Module1 module from MyLib and rename it to IniFile.
11. Choose the Build toolbar button.



You have now created a DLL; and it is now ready to be used in the South Seas Adventure application.

Using a DLL

At this point, you have created a DLL—although no .DLL file has been generated (a step which comes later in this lesson). You can still, however, use the DLL stored in the repository in the same way that you used the MyLib shared library:

1. From the Repository Explorer, select the South Seas Adventures application by double-clicking its branch.
2. Choose the Application Properties toolbar button.
The Properties dialog box displays.
3. Select the Libraries tab.
4. Scroll through the Included list box and remove MyLib by double-clicking it.
5. Go through the Available list box and select MyDLL by double-clicking it.
6. Choose OK.
7. Choose the Build toolbar button to recompile the entire application.



The South Seas Adventures application is now ready to be executed and will run just as before. However, where it used to access code from the library, MyLib, it now accesses code from the DLL library, MyDLL. A few more steps will be necessary to create and use an external .DLL file.

Creating a .DLL File

To use a CA-Visual Objects .DLL file, you must do the following:

- Create the physical .DLL file
- Create a library defining the public protocol for the DLL
- Include the public protocol library in the search path of the application

Let's first create the .DLL file:



1. From the Repository Explorer, select MyDLL by clicking its branch.
2. Choose the Application Properties toolbar button.

The Application Properties dialog box appears.

3. Make sure you have the full path for the .DLL file as %CavoSamplesRootDir%\SSATutor, then choose OK.
4. Select the Make DLL command from the Application menu.

This creates two files: MYDLL.DLL and MYDLL - DLL.AEF. The .DLL file contains the executable code and the .AEF file contains the prototypes that you will use to access the DLL.

Both files are written to the path for the .DLL file, as specified in the Application Properties dialog box.

Tip: At this point, the .DLL and .AEF files can be distributed to other CA-Visual Objects programmers.

Using a CA-Visual Objects .DLL

To have your application access the code stored in a .DLL file, you must define the prototypes to the entities you want to access, similar to the way Win32 API library contains the prototypes to the functions available in Windows.

For .DLLs that are created in CA-Visual Objects, this prototype library is created for you when you generate the .DLL file. All you have to do is import the generated .AEF file and include it in the application's search path.

Importing Your
.AEF file

Let's import the sample MYDLL - DLL.AEF file:

1. Select the Default Project in the Repository Explorer.
1. Select the Import command from the File menu.
2. Select MYDLL - DLL.AEF from the SAMPLES\SSATUTOR subdirectory of your CA-Visual Objects 2.7 directory and click Open.

A new application is added—MyDLL.DLL.

The new application is a shared library. It contains only the prototypes of the entities in MYDLL.DLL.

Tip: If there are entities in the DLL that are not intended for direct use by the programmer, their prototypes can be removed from the library. You would then re-export the library as an .AEF file for distribution.

Including Your .DLL
in an Application

Finally, to use this new .DLL file, you must include the library which contains its prototypes in your application search path.

Let's include your new library in the South Seas Adventures application:

1. From the Repository Explorer, select the South Seas Adventures application by clicking its branch.
2. Choose the Application Properties toolbar button.



The Properties dialog box appears.

3. Select the Libraries tab.
4. Go through the Included list box and remove MyDLL by double-clicking it.
5. Go through the Available list box and select MyDLL DLL by double-clicking it.
6. Choose OK.
7. Select the Build toolbar button to rebuild the revised South Seas Adventures application, which includes the new library.



From now on, South Seas Adventures will use the external .DLL file and not the DLL entry in your repository. Therefore, any change to the repository source code must be followed by generating the .DLL/.AEF pair and importing the .AEF library once again. Otherwise, the dependent application will not recognize the change.

Summary

In this lesson, you have learned how to use libraries and dynamic link libraries to make your applications more efficient. First, you learned to share code between applications by using libraries. You accomplished this by moving a module's code into a shared library and making use of it in an application. The next step was to link this library into your application.

Finally, you learned how to create DLLs so that your code is shared at runtime. You also learned how to create and use an external .DLL file so that your code can be developed for the most flexibility and portability.

In the next lesson you will create installation disks and learn how to use the Install Maker and CA-Installer for the proper distribution of your CA-Visual Objects 2.7 applications.

Distributing Your Application

When you complete this lesson, you will be able to create a set of installation disks for your CA-Visual Objects 2.7 application for final distribution.

Overview

The final stage of the development cycle involves distributing your application. CA-Visual Objects provides two utilities, Install Maker and CA-Installer, that allow you to distribute your applications in a timely and professional manner.

Install Maker

Your CA-Visual Objects applications are made up of many files (for example, .EXE, .DLL, and data files). Install Maker helps you determine what files are necessary to run the application and allows you to quickly create disk images for the application.

Install Maker automates part of the file gathering process. By using information which is stored in the repository, it can deduce which files should be included on the installation disks (system DLLs, RDDs, and so on). You only need to specify the files you create—such as data files, report files, and help files.

Install Maker also allows you to specify information for use at installation time, such as:

- The default installation (or target) directory
- The name of the Start Menu folder that will contain your application

This information is then used by the CA-Installer program.

CA-Installer

As part of the disk generation process, Install Maker includes the CA-Installer program (SETUP.EXE) on your disk set or CD-ROM. This is the program your users run to install the application. It allows your program to be installed in much the same way as CA-Visual Objects 2.7.

Exercise

In the following exercise, you have the opportunity to create the distribution disks for the South Seas Adventures application using Install Maker. You can then install the application from these disks using CA-Installer.

Generating the Executable

Before you begin this process, you must create the application's executable file:



1. Select the South Seas Adventures application from the Repository Explorer.
2. Choose the Make EXE toolbar button.
3. After the .EXE file is generated, close CA-Visual Objects 2.7 by double-clicking its system menu.

Note: Install Maker cannot run if CA-Visual Objects 2.7 is open.

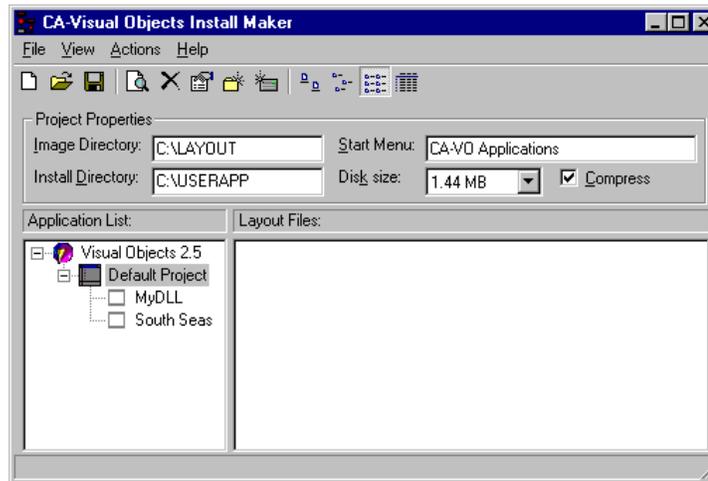
Using the Install Maker

Once the executable file is generated, you can prepare the distribution disks for the South Seas Adventures application:



1. Start the Install Maker by selecting it from the CA-Visual Objects 2.7 Start menu folder.

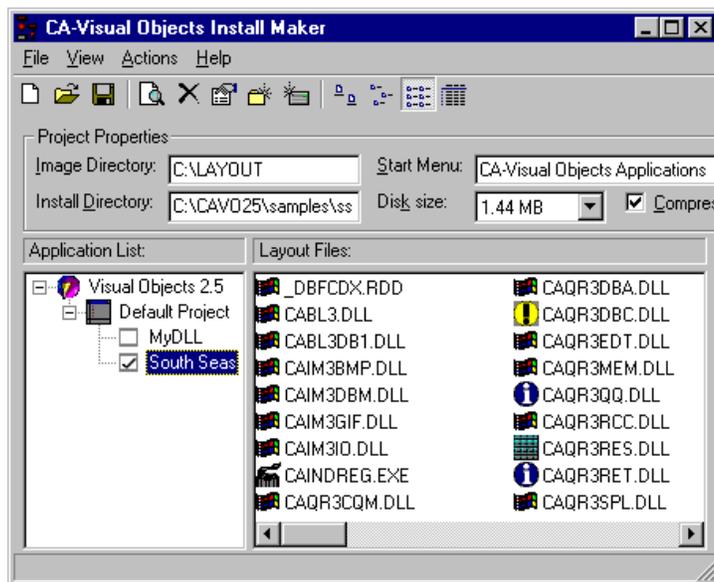
The CA-Visual Objects Install Maker dialog box appears:



The Application List list box displays all applications and DLLs that are contained in your CA-Visual Objects 2.7 repository.

2. Select South Seas from the Application List. This will place a check mark in the box on the tree.

Using information from the repository, Install Maker populates the Layout Files list box with the system file names associated with this application, as well as the application's executable (in this case SSA.EXE):



If your application is made of many executable and/or user defined DLLs, you can specify more than one application. The South Seas Adventures application is now using the DLL you created in a previous lesson.

3. Select MyDLL in the Application List and a check mark will also be placed on the tree next to MyDLL. This includes the DLL file you created with the other layout files.

Disk Images

Instead of creating disks directly, Install Maker creates *disk images* on your hard drive. Each disk image resides in a subdirectory (DISK1, DISK2, and so on) under the specified directory, which Install Maker populates with the files required for installation:

1. To change the location of these disk images, select the Image Directory edit control.
2. Type in a path name on your hard drive (for example, C:\CAVO27\SAMPLES\SSATUTOR\LAYOUT).

Disk Size

You must also specify the disk size you are planning to use for installation disks. To change the disk size, select the Disk Size drop-down list box. In this lesson, we use the 1.44 MB disk size.

Note: The Make Disks process, as described later, can be run for each disk format you require.

Program Manager Group

When your program is installed, you can specify a default Start Menu folder to be created for your application by following these steps:

1. Select the existing text in the Start Menu edit control.
2. Type South Seas Adventures.

Default Installation Directory

When your program is being installed, CA-Installer will prompt the user for a default drive and directory for installation and suggest your specification. To specify the target installation directory:

1. Select the existing text in the Default Install Directory edit control.
2. Type **C:\SSA**.

The C: drive is the most common location. As for the directory you specify, if it does not exist, it is created on your user's drive by the CA-Installer program. Your user also has the option of specifying a different directory.

Specifying Other Files for Installation

When you select the South Seas Adventures application and MyDLL to be included in the installation, most of the related files are automatically included in the Layout Files list box. For example, the executable (.EXE) file for South Seas Adventures, the .DLL file for MyDLL, and several other .DLL files are included for installation.

Other files needed to run the application, such as data files (.DBF), must be manually added to the list box.

Note: For this application, index files do not have to be included since they will be generated the first time the installed application is run.

Let's add the additional files needed for the South Seas Adventures application from the CA-Visual Objects 2.7\SAMPLES\SSATUTOR subdirectory, as follows:

1. Choose the Add command from the Actions menu.

You are prompted with a standard Open dialog box.

2. Select the following files from the SAMPLES\SSATUTOR subdirectory while pressing the CTRL key, then choose Open:

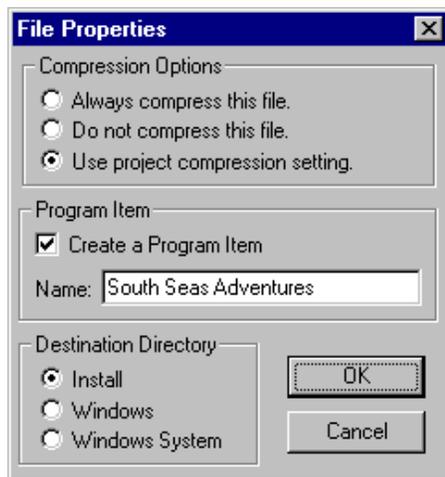
ACCINVC.DBF	INVHDR.DBF
ACCPAY.DBF	INVNRPT.RET
ADVDTL.DBF	ITEM.DBF
ADVHDR.DBF	OUTSTPAY.RET
CADVRPT.RET	PAYMENT.DBF
CUSTLIST.RET	PAYRPT.RET
CUSTOMER.DBF	SSA.HLP
CUSTOMER.DBT	STATE.DBF
EMPLOYEE.DBF	SYSKEY.DBF
INVCRT.RET	TENDER.DBF
INVDTL.DBF	

Specifying File Properties

There are a number of properties that you can specify for each file in the Layout list. Let's take a look at these:

1. Scroll through the list of files until you find South Seas Adventures, SouthSeas.EXE.
2. Right-click on it and choose the Properties command.

The Properties dialog box appears:



Destination Directory

From this dialog box, you can specify whether the selected file is to reside in the Install directory, the Windows directory, or Windows System directory. By default, your application executable file (SSA.EXE) is set to reside in the install directory.

Program Item and
Program Item Name

The Program Item group box allows you to specify whether the file appears as a Program Item in the South Seas Adventures Start Menu folder. By default, your application is set to appear as a Program Item.

You can also change the name of the Program Item to what you want it to appear as in the Program Folder, since Install Maker defaults to the file name minus the extension. You can set the Program Item name as follows:

1. Select the Program Item Name edit control and replace SouthSeas with **South Seas Adventures**.
2. Choose OK to accept the changes.

Although Windows applications require many files, typically only a few appear on their Start Menu folder. For example, the South Seas Adventures application, as far as your user is concerned, is made up of the executable and the help files.
3. To change the file properties for the South Seas Adventures help file, select SSA.HLP from the Layout Files list box and right-click, then choose Properties from the local pop-up menu.
4. Enable the Program Item check box.
5. In the Program Item Name edit control, type **South Seas Help**.
6. Choose OK to accept the changes.
7. Select MyDLL.DLL from the Layout Files list box, right-click it, then choose Properties.
8. Disable the Program Item check box.
9. Choose OK to accept the changes.

Creating a Project File

It is advisable to create a project file that stores, among other things, the list of files in the Layout Files list box. Let's create the project (.PRJ) file as follows:

1. Select the Save As command from the File menu.
2. Change the drive and directory to that of your CA-Visual Objects 2.7 system directory (for example, C:\CAVO27).
3. In the File Name edit control, type **SSA01.PRJ** and choose OK to save the file.

If you need to create another set of disks, use the Open command from the File menu to load all the information from this project file (SSA01.PRJ). If you make any modifications, you can then you can use Save As to create additional project files (such as SSA02.PRJ, SSA03.PRJ, etc.).

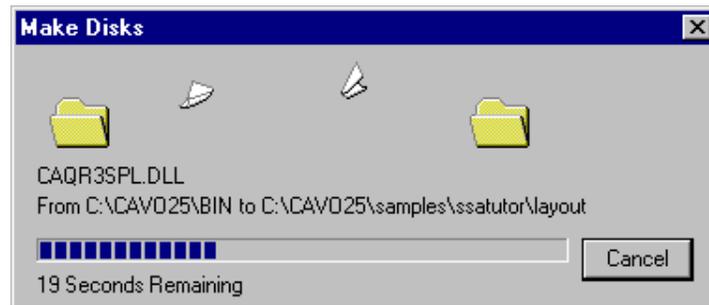
Now that you have saved the project information, you can create the disk images.

Creating Disk Images

Once you have defined all of the files required to run your application, you are ready to create the disk images:

1. Choose the Make Disks command from the Actions menu.

While CA-Visual Objects 2.7 is preparing your files, the Make Disks dialog box displays.



2. Once the process is completed, Choose Exit from the File menu to shut down Install Maker.

Creating and Testing the Distribution Disks

The final steps are creating distribution disks from your disk images and testing the installation process to ensure you have not forgotten any files:

1. Create your disks by copying the contents of each DISK subdirectory to a separate diskette and labeling the diskette accordingly. The Windows Explorer can be used to accomplish this task.
2. After copying all disk images, insert DISK1 into drive A: (or B:).
3. From the Start menu, choose the Run command.
4. Type **A:\SETUP.EXE** (or **B:\SETUP.EXE**) in the Command Line edit control and choose OK.
5. Once the program is installed, you can test the application by double-clicking its program item in the South Seas Adventures Start Menu folder.

Important! To ensure proper testing of the installed executable version of your application, you should test it on a computer that does not have CA-Visual Objects 2.7 installed. Your testing at this phase should be at least as rigorous as the testing you performed on the application under dynamic execution.

Summary

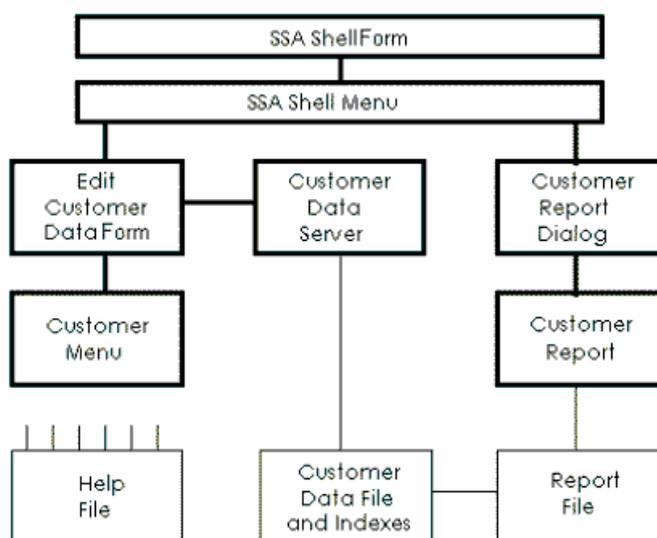
In this lesson, you learned how to use CA-Visual Objects 2.7 Install Maker utility to prepare your application for distribution.

Congratulations! You have now successfully completed all of the lessons for the South Seas Adventures tutorial. The information that you have learned will help you to create, manipulate, and distribute your own CA-Visual Objects 2.7 applications.

Creating a Path-Independent Application

This appendix describes what you must do to make sure that an application can run successfully when it is installed on any drive or directory. Creating such path-independent applications is an important design objective.

The following diagram of the primary South Seas Adventures application building blocks indicates that there are three types of external files—help files, data/index files, and Computer Associates Report Editor report definition files:



Note: Rectangles with thick borders are the primary building blocks, while those with thin borders are external files. Design linkages are shown as thick lines, while external linkages are shown with thin lines.

An application must be able to locate these files at runtime, so there are several key steps that must be taken to remove possible path dependencies while you are creating the building blocks of your application. Report files themselves contain information that directs the report runtime engine to the location of the related data files, so additional steps to remove path information from each report's query statement are required.

Establishing Drive and Directory Independence

With regards to carefully planning the directory structure for the developer's development environment, it is important to think ahead to the drive and directory possibilities when the application is installed by an end user.

There are two possible approaches when planning your directory structure, depending on where you want your data to reside.

Fixed Paths

If you are sure that the data files will always reside on a given drive and directory, then you may want to use fixed paths for the data file and index files specified in the DB Server Editor. This may be a good choice if the data files are maintained on a single LAN drive and directory. During the development process, you can keep your test data files in these designated locations.

User-Defined Paths

If the location of the data files depends upon the user's choice of installed directory, you should plan to keep your application help files, data files, and report files in the application directory (that is, where the .EXE file resides). In this case, you must not use any path information when you define each DATA server and report. During the development process, these files should be kept in the application's .EXE file directory (for example, D:\CAVO27\SAMPLES\SSATUTOR) specified in the Application Properties dialog box.

This appendix addresses the second approach—since this is the more likely scenario—which places some restrictions on what you can do. The fact that the application starts up from the directory that holds the .EXE file allows you to specify path-independent linkages for your external files.

There are several runtime considerations relating to finding these external files:

- When running an executable file, the current directory is the directory that holds the .EXE file. External files that have no explicit path specified in the related application building blocks will be successfully located if they are placed in this directory.
- When you use Save or Save As to store a report definition, the path is stored in the report's class entity. This path is required during the development process. However, you can create a special access entity to change this path to NULL_STRING at runtime, allowing report files stored in the application directory to be successfully opened.
- Each report file executes a query at runtime to locate the data files and index files. One or more data servers must be specified when you define a new report and any path information stored in the data server at that time becomes part of the query statement.

- Achieving drive and directory independence for your reports requires that you not use any path information in defining data servers. Otherwise, the runtime query executed for each report will fail if the data files are not in the fully specified location. The South Seas Adventures application was designed to achieve drive and directory independence. Therefore, there is no path information associated with any data server.

Help Files

You can specify a help file name for a shell window or a data window by using the Help File Name property that is specified in the Window Editor. The Window Editor generates source code that creates the linkage. For example, if the help file is SSA.HLP, the code is:

```
SELF:HelpDisplay := HelpDisplay{"ssa.hlp"}
```

Where SELF is a shell window or data window. Here, a help display object is created using the SSA.HLP file, and then using the HelpDisplay assign of the Window class. This help linkage is path-independent.

If we had included path information in the Help File Name property, then it would be included in the source code. This would require that the help file always be in the designated location, or that the path is modified at runtime. Generally, you can simply remove all path information from the Help File Name property and locate the help file in the directory in which the application is installed.

DB Server Data Files

If you create a data server by importing a .DBF file, path information will automatically be placed in the File Name edit control. If you do not remove the path information from the server before you save it, the path will be stored in the CLASS definition, as follows:

```
CLASS Customer INHERIT DBServer
INSTANCE CDBFPath := ;
    "c:\cavo27\samples\ssatutor\" AS STRING
```

Therefore, you must remove any path information for the data file or any index file, leaving only the name of the file. If you do this, NULL_STRING is stored in the instance variable, as shown below:

```
INSTANCE CDBFPath := "" AS STRING
```

If, for some reason, you do want path information in the CLASS definition, you can still change it at runtime by creating a special access method for the instance variable. This access entity removes path dependence at runtime:

```
ACCESS CDBFPath CLASS Customer
RETURN ""
```

This next code allows you to specify the DBFS subdirectory of the current directory:

```
ACCESS CDBFPath CLASS Customer
RETURN ".\dbfs\"
```

However, neither of these access methods address the problem of path independence for reports, nor does use of the SetDefault() function, which allows you to specify a path that will be searched when performing direct database actions.

Important! To achieve path independence in reports, you must not have any path information stored in any data server building blocks.

Report Files

Report Editor File Path When you create a report you will use either the Save or Save As command in the Report Editor. If you are creating a new report, the default location is your application .EXE file directory; therefore, you must use Save As to locate it elsewhere. The actual path location of the file is stored in the CLASS entity as follows:

```
CLASS CAdvRpt INHERIT ReportQueue
INSTANCE CAdvRpt_File := ;
"%Cavo27SamplesRootDir%\ssatutor\cadvrpt.ret" AS STRING
```

This path is required so that the file can be opened when you want to edit the report. Thus, you must use a special access method to achieve path independence in the end-user runtime environment. The South Seas Adventures application includes an access that removes the path dependency for each report. For example:

```
ACCESS CAdvRpt_File CLASS CAdvRpt
RETURN "cadvrpt.ret"
```

While this access method achieves path independence at runtime, you must also address the location of the report file during the development process. The Report Editor file path in the CLASS statement must be changed, if you have imported the South Seas Adventures .AEF file from a directory other than the SSATutor subdirectory and you wish to edit the report definition.

You can change the CLASS statement by entering the Report Editor and using Save As to save it to the desired location. When you double-click on the report entity for the first time, a dialog box asks you to type in the full path of the .RET file. When you do this and close the dialog box, the Report Editor opens. The path in the CLASS definition is updated to the new path.

Query Path

When you create a new report, the Report Editor dialog box requires that you choose one or more servers to define what fields are on the report. If any of these servers contain path information for a data or index file, it will be included in the query statement.

In order to achieve path independence, you must manually remove the paths by using the Database Edit Query command. After doing this, save the Report Editor file to the proper location. If you do not manually remove path information from the query, you will get a Report Editor runtime error if the data files cannot be found by using the designated path.

Important! *The best approach is to remove the path information from the data servers so that it is not passed to the Report Editor.*

Icon, Cursor, and Bitmap Files

Icons, cursors, and bitmaps can be treated differently than help files, data files, and report files, since they are directly incorporated in any .AEF export file or a generated .EXE file. When the South Seas Adventures application was initially created, the RESOURCE definitions of the icons, cursors, and bitmaps contained the path for the .ICO file on the developer's disk drive. For example:

```
RESOURCE ITEM_ICON Icon ;
    %Cavo2\SamplesRootDir %\ssatutor\files
```

If you have installed CA-Visual Objects 2.7 to the CAVO27 directory, the file is stored in this directory during the .AEF file import process. This is not a problem because CA-Visual Objects 2.7 automatically modifies the path during the import process. The file is placed into the directory from which the import is taking place and the RESOURCE statements will be automatically changed. For example, if you installed to drive D:, the icon, cursor, and bitmap files will be placed in the D:\CAVO27\SAMPLES\S ATUTOR\FILES subdirectory.

Because this is automatically handled during the import process, no further steps are necessary unless you wish to edit an icon or cursor itself. If you try to edit such an icon entity and it is not in the same location as the original .AEF file, a dialog box is displayed, indicating that the file cannot be found. You must close this dialog box in order to invoke the Icon Editor.

When in the Image Editor, use the File Open command to open the file from its new location. Then, use the Save command, type in the same entity name, and choose OK to save the icon file and the icon entity. Select Yes when asked about overwriting the existing file. If you are working with a cursor entity, you must first change to the Cursor Mode (from the Options menu) when you invoke the Image Editor.

Summary

As you have just seen, there are essentially no runtime issues associated with the paths for icons, cursors, and bitmaps, since they are bundled into the .EXE file. The only issue is one of moving an .AEF file between two developer's machines with different directory structures.

In addition, you have learned what you can do to make your help, data, index, and report files path-independent.

Index

A

Accessing and Updating Data

- access and assign methods, 140
- controls, 144
- data forms, 143
- data servers attached to data forms, 144
- DBServer class, 142
- generated data server classes, 141
- overview, 139
- SQL class, 142
- using a method, 140
- using access and assign methods, 141
- using protected variables, 140
- Xbase compatibility, 139

Adding Controls to Your Windows

- check box controls, 102
- combo box controls, 99
- fixed icon controls, 108
- list box controls, 105
- multiline edit (MLE) controls, 98
- overview, 95
- push button controls, 109
- radio button and radio button group controls, 102
- single-line (SLE) controls, 96
- single-line edit controls as data servers, 96

Adding Help to Your Applications

- assigning help to a menu command, 219
- assigning help to a window, 218
- assigning help to controls, 219
- attaching your help file, 216
- creating help files, 224
- F1, 215
- help context property, 218
- HelpRequest event system, 217
- HelpRequest(), 217
- implementing context-sensitive, 216
- implementing direct calls to help
 - menu commands, 223

- overview, 223
- invoking context-sensitive, 215
- invoking context-sensitive help, 220
- invoking WinHelp, 217
- overview, 215
- shift+F1, 216
- viewing help for a control, 221
- viewing help for a menu command, 222
- viewing help for a window, 221

App:Exec() method, 18

App:Quit() method, 18

Application Building Blocks, 15

Application design

- building blocks, 19
- cursors, 23
- data fields, 20
- data servers, 20
- data tables, 19
- data window
 - brower window, 20
 - overview, 20
- data-aware controls, 21
- detail subform, 20
- developer-coded entities, 25
- draw objects, 23
- edit window, 20
- entity types, 25
- event handlers, 22
- event-oriented methods, 27
- help systems, 23
- icons, 23
- linking other building blocks, 26
- linking primary building blocks, 24
- master-detail edit window, 20
- menus, 21
- new window, 20
- overview, 19
- reports, 22
- shell and dialog windows, 21
- subform window, 20
- system-generated entities, 26
- view window, 20

- window controls, 21
- Application details
 - code files, 31
 - compiler options, 31
 - components, 31
 - dependency information, 31
 - directory information, 31
- Application environment
 - application options, 38
 - compiler options, 39
 - configuring, 38
- Application modules, viewing, 40
- Auto Layout
 - menus and toolbars, 121
 - windows, 90
- Automated Make and Entity-Level Compiling, 32

B

- Bitmaps
 - creating, 176
 - declaring as a resource, 175
 - declaring as a subclass, 176
 - instantiating an `SSABitmap` object, 176
 - overview, 175
 - `SSABitmap`, 176
- Brower window, 20
- Browse view mode, 80

C

- `CDecimal()` function, 151
- Check box controls
 - overview, 102
- Child application windows, 79
- Child servers, 63
- Client data forms, 62
- Combo box controls
 - creating, 99
 - invalid code entry disabled, 101
 - overview, 99
- Compiling and testing changes, 93

- Component entity level, 32
- Component heirarchy
 - application level, 32
- Component module level, 32
- Context-sensitive help
 - definition, 215
 - implementation, 216
- Control order and multiple groups, 112
- Controls, dynamic positioning, 179
- Creating a customer data server
 - importing a `.DBF` File, 49
 - importing an index, 51
 - invoking the `DB Server Editor`, 47
 - overview, 47
 - saving the data server, 53
- Creating a data form
 - adding a push button, 91
 - Auto Layout, 90
 - compiling and testing your changes, 93
 - creating a data window template, 88
 - designing window layout, 90
 - importing a support module, 88
 - overview, 87
- Creating a modal dialog box
 - overview, 84
 - retrieving values, 85
 - warning box modal dialog forms, 84
- Creating and using windows, overview, 77
- Creating Menus and Toolbars
 - attaching a menu to a data window, 133
 - changing toolbar position, 130
 - checking a menu item, 127
 - collapsing/expanding the menu structure, 123
 - creating a new menu, 120
 - creating a new module, 119
 - creating a toolbar, 128
 - designing a menu, 135
 - hierarchy
 - adding items, 123
 - changing menu items, 124
 - removing menu items, 125
 - menu shortcuts, 126
 - other modifications, 132
 - overview, 119
 - previewing menus, 122
 - putting it all together, 134
 - saving menus, 132
 - specifying menu actions, 125
 - `SSAWindow` event name, 126

- toolbar spacing, 131
- using Auto Layout, 121

Cursor

- creating and modifying, 166
- importing, 167
- predefined, 166

Customizing generated code, 117

Customizing Window Event Handlers

- EditChange() method, 150
- event methods, 149
- NewPaymentWindow, 149
- Notify(), 153
- overview, 147
- Queryclose event handler, 155
- viewing your results, 152

CWhole() function, 151

D

Data forms

- data propagation, 80
- form and browse view modes, 80
- overview, 80
- server use, 80

Data propagation, 80

Data servers

- attaching a field spec, 72
- creating a customer data server, 47
- methods compatible with the Xbase DML, 46
- overview, 45
- planning data server field properties, 71
- saving, 53

Data tables, 19

Data validation, 81

Data Validation

- disconnected controls, 81
- overview, 81
- using the Window Editor, 81

Data window, 20

Data-aware controls, 21

DataDialog forms, 79

DB Server Editor

- description, 46
- invoking, 47

Debugger, 205

Debugging Your Application

- correcting the error, 212
- Debugger, 205
 - introducing an error, 207
 - viewing the error, 206
- Error Browser, 201
 - imported modules with errors, 202
 - resolving errors, 203
- evaluating the expression, 211
- locating the bug, 210
- overview, 201
- running the application with the Debugger, 209
- setting debug on
 - at the module level, 208
 - overview, 208

Defining Field Specifications

- attaching a field spec to a data server field, 72
- creating and modifying, 70
- creating field specs from the DB Server Editor, 74
- overview, 69
- planning data server field properties, 71

Detail subform, 20

Dialog forms, 78

Dialog windows, 21

Directory structure

- overview, 29
- SSATUTOR, 30, 31

Distributing Your Application

- CA-Installer, 241
- creating a project file, 246
- creating and testing distribution disks, 247
- creating disk images, 247
- default installation directory, 244
- default Start Menu folder, 244
- destination directory, 245
- generating an executable, 242
- Install Maker
 - disk images, 243
 - disk size, 243
 - overview, 241
- Intall Maker
 - using, 242
- overview, 241
- Program Item, 246
- Program Item name, 246
- specifying file properties, 245
- specifying other files for installation, 244

DLL

- copying modules between application, 236

- creating a .DLL file, 238
- creating a new DLL application, 236
- including in an application, 239
- overview, 236
- using, 237
- using a CA-Visual Objects .DLL, 238

E

Edit window, 20

EditChange(), 150

Entity Modules

- Adventure:Data, 33
- Adventure:Forms, 34
- Adventure:Methods, 34
- App:Misc, 34
- App:Reources, 34
- App:Start, 34
- design considerations, 32
- naming conventions, 33
- Password:Forms, 34
- SSAChild:Menu, 34
- SSAShell:Forms, 34
- SSAShell:Menu, 34

Error Browser, 201

Event handler methods, 27

Event handlers, 22

Event notification

- broadcast message activation, 65
- child servers, 63
- client data forms, 62
- manual, 64
- overview, 62

Event-oriented methods, 27

Executable, 242

F

FieldSpec

- attaching to a data server field, 72
- creating, 71
- creating from the DB Server, 74
- invoking the editor, 70
- properties, 71

Fixed icon controls, 108

Form view mode, 80

H

Help

- creating, 224
- project files, 224
- topic files, 224

Help systems, 23

I

Icon

- attaching to shell forms, 164
- creating, 157
- displaying on a window, 165
- in the Program Group, 162
- labeling your application, 162
- saving, 160

Importing

- applications, 37
- DBF File, 49
- index, 51
- support module, 60

Inheritance and subclassing

- creating a subclass, 116
- customizing generated code, 117
- overview, 115
- parent, 115
- subclass, 115
- subclassing with generted code, 116
- superclass, 115

Installing ODBC drivers, 54

Integrated Development Environment

- IDE, 15
- incremental development, 15
- multi-tiered repository, 15

L

Libraries and Dynamic Link Libraries
libraries, 231

Libraries and Dynamic Link Libraries
application maintenance, 232

- building a library, 235
- creating a new library application, 233
- dynamic link libraries, 231
- efficiency, 232
- library distribution, 232
- moving modules between applications, 234
- overview, 231
- using a library, 235

List box controls, 105

M

Manual event notification, 64

Master-detail edit window, 20

MDI

- child application windows, 79
- DataDialog forms, 79
- dialog forms, 78
- modal dialog forms, 79
- modeless dialog windows, 79
- overview, 78
- shell forms, 78

Menu

- attaching to a data window, 133
- checking an item, 127
- collapsing/expanding the menu structure, 123
- creating, 120
- designing, 135
- hierarchy
 - adding items, 123
 - changing menu items, 124
 - removing menu items, 125
- Menu Editor, 21
- previewing, 122
- saving, 132
- shortcuts, 126
- specifying actions, 125

Menu event methods, 27

Modal dialog forms, 79

Modeless dialog windows, 79

Module, creating, 119

Multiline edit (MLE) controls

- moving the MLE control, 98
- overview, 98
- viewing your results, 98

Multi-tiered repository, 15, 31

N

Name-based linkages, 80

naming controls, 112

New window, 20

Notify()

- creating the method, 153
- overview, 153
- viewing your results, 154

O

o, 18

Objects

- App:Exec()method, 18
- App:Quit()method, 18
- braces {}, 18
- creating, 18
- defining, 17
- description, 16
- Init() method, 17
- interaction, 17
- methods, 17
- runtime creation and destruction, 17
- used in South Seas Adventures, 18
- working with, 18

ODBC

- administrator, 55
- installing drivers, 54

OpeningDialogResize(), 174

P

Path-independent application

- DB Sever data files, 251
- drive and directory independence, 250
- help files, 251
- icon, cursor, and bitmap files, 253
- overview, 249
- query path, 253
- Report Editor path, 252

Path-independent applications, creating, 31

Programming techniques

- control order and multiple groups, 112

- naming controls, 112
- overview, 110
- Tab and group stops, 110

Programming with servers

- event notification, 62
 - broadcast message activation, 65
 - child servers, 63
 - client data forms, 62
 - manual, 64
- importing a support module, 60
- overview, 60
- running the application, 62
- Viewing the Server Source Code, 61

Push button

- adding, 91
- controls, 109
- methods, 27

R

- Radio button and radio button group controls, 102

Report Editor

- adding fields, 189
- customization, 189
- header and footer, 191
- overview, 182
- parameters, 193
- passing parameters, 196
- quick tour, 188
- running within your application, 192
- saving your work, 192
- using the Report Editor, 182
- verifying results, 198

Reports, 22

Repository Explorer, using, 35

Retrieving values, modal dialog forms, 85

Running applications, 62

S

SDI

- overview, 77
- top application windows, 77

SetSelectiveRelation, 64

Shell forms, 78

Shell windows, 21

Single document interface (SDI), 77

Single-line edit (SLE) controls

- overview, 96
- single-line edit controls as data servers, 96

Source code entities

- assign, 17
- methods, 17

South Seas Adventures

- application building blocks
 - overview, 16
- application design, 19
- IDE, 14
- importing the application, 37
- MDI, 14
- objects, 16
- overview, 13
- system operations covered, 13
- you should know, 16

SQL Editor, 58

SQL Server, creating, 53

Subform window, 20

T

- Tab and group stops, 110

Text objects, 178

Toolbar

- changing position, 130
- creating, 128
- other modifications, 132
- spacing, 131

V

View window, 20

Viewing a MDI Application

- adding functionality, 82
- overview, 82
- shell form, 82

Viewing server source code, 61

W

Warning box, modal form, 84

Win32 API Functions

- calling, 227
- definition, 227
- overview, 227
- windows metric information, 228

Window controls, 21

Windows Metric Information, retrieving, 228

Working with Draw Objects

bitmaps

- creating a bitmap object, 176
- declaring as a resource, 175
- declaring as a subclass, 176
- overview, 175
- SSABitmap, 176

Bitmaps

- instantiating an SSABitmap, 176
- dynamic positioning of controls, 179
- making the dialog box resizable, 173
- overview, 171
- text objects, 178
- the resize event, 174
- viewing the results in the application, 180

Working with Icons and Cursors

cursor

- creating and modifying, 166
- importing, 167
- predefined, 166

icon

- attaching to shell forms, 164
- creating, 157
- displaying on a window, 165
- in the Program Group, 162
- labeling your application, 162
- saving, 160

overview, 157

