Visual Objects

For
Windows 2000® and Windows XP®

Getting Started Guide
Version 2.7

Contents

Chapter 1: Introduction

Welcome to Visual Objects! 1–1
- Visual Programming Tools and a Complete IDE 1–2
- A Fully Object-Oriented Language 1–3
- Open Database Access 1–4
- An Active Repository 1–6
- A Native Code, Incremental Compiler 1–6
- A Portable Executable Format, Incremental Linker 1–7
- Reporting Using the Report Editor 1–8
- An Open Architecture 1–8

Visual Objects Features 1–9
In This Guide 1–11
What You Need to Know 1–13
General Typographic Conventions 1–13
Getting Help 1–15

Chapter 2: Installing and Starting Visual Objects

Installing Visual Objects 2–1
AutoStart Installation 2–1
Manual Installation 2–1
Before Starting Visual Objects 2–3
Starting Visual Objects 2–4
What's Next 2–5

Chapter 3: Object-Oriented Programming Concepts

Why Object-Orientation? 3–1
The Paradigm Shift 3–2
What Is an Object? 3–4
What Is a Class? 3–5
Inheritance: Superclasses and Subclasses 3–9
A Real-World Example 3–12
Additional Strengths of OOP 3–15
Encapsulation 3–15
Modularity and Reusability 3–16
Summary 3–17
The Visual Objects Libraries 3–19
What's Next 3–22

Chapter 4: An Overview of the IDE

Repository-Based Development 4–1
The IDE Tools 4–4
The Repository Explorer 4–6
Managing Projects 4–8
Browsing Applications and Modules 4–9
Viewing Entities at the Module Level 4–10
Viewing Entities at the Entity Level 4–12
Browsing Classes 4–14
Error Browser 4–18
The Editors 4–20
Source Code Editor 4–22
Data Server Editors 4–24
The FieldSpec Editor 4–25
Window Editor 4–26
Menu Editor 4–38
Report Editor 4–39
Image Editor 4–41
The Debugger 4–41
What's Next 3–43

Chapter 5: Learning the Basics

Lesson 1: A Tour of the Standard Application 5–3
Creating an Application: Using the Application Wizard 5–4
Building and Running the Standard Application 5–5
A Closer Look at the Application 5–6
MDI Application Structure 5–7
The App:Start() Method 5–9
The Shell Window 5–10
The Empty Shell Window 5–15
The Standard Shell Window 5–17
The Child Windows 5–22
Default Event and Error Handling 5–24

A Closer Look at the Standard Application 5–26
The Empty Shell Window 5–26
Opening Database Files 5–28
Switching Between Form and Browse View 5–30
The Standard Shell Window 5–33
Opening Multiple Windows 5–34
Using OLE Database Files 5–36
Summary 5–38
Lesson 2: Setting Up the Data Servers 5–40
Importing the OE Data Servers Library 5–41
A Quick Tour of the Customer Data Server 5–43
Loading the Customer Data Server 5–43
The Indexes List Box 5–46
The Fields Group Box 5–49
Creating the Orders Data Server 5–52
Starting the DB Server Editor 5–52
Importing the Database File 5–54
Importing the Index Files 5–56
Browsing Data 5–58
Sharing Field Specifications 5–59
Customizing Field Properties 5–62
The FieldSpec Editor 5–64
The Source Code 5–68
Building the OE Data Servers Library 5–70
Adding the Library to Order Entry’s Search Path 5–70
Summary 5–71
Lesson 3: Creating a Data Window 5–72
Starting the Window Editor 5–73
Window Properties 5–75
Using Auto Layout 5–77
A Closer Look at the Main Data Form 5–80
A Closer Look at the Sub-Data Form 5–82
Customizing Windows 5–85
Viewing Tab Order 5–87
Moving On 5–88

The Source Code . . . 5–89
Summary . . . 5–90
Lesson 4: Modifying the Menu . . . 5–91
Adding the Customer Orders Menu Command . . . 5–92
Previewing Your Work . . . 5–94
Summary . . . 5–94
Lesson 5: Adding the Ordering Methods . . . 5–95
Modifying the Menu . . . 5–96
Adding Commands . . . 5–96
Defining Menu Properties . . . 5–99
Creating the Methods . . . 5–101
The TEXTBLOCK Entity . . . 5–102
The Methods . . . 5–103
Enabling the Menu Commands . . . 5–105
Summary . . . 5–107
Lesson 6: Running the Order Entry Application . . . 5–108
Generating an Executable File . . . 5–108
Running the Application . . . 5–109
Looking at the New Features . . . 5–109
What’s Next . . . 5–115

Chapter 1

Introduction

Welcome to Visual Objects!

Visual Objects is a fully object-oriented application development system that allows you to quickly and easily create sophisticated applications that run under Microsoft Windows and Windows NT. Its power and flexibility offer new opportunities and technology to application developers of all levels and backgrounds.

Visual Objects gives you the power to create high-performance, mission-critical, cutting-edge applications and components that deliver everything Windows users have come to expect, including:

- Multiple document interfaces (MDI), with no constraints on simultaneously opening several documents (such as databases or text files) or the same document in several different windows
- Event-driven operation, with no limitations on user flexibility and control
- Top-flight graphical appearance (including support for Windows common controls, OCX controls, and OLE 2.0) and full-fledged annotation, prompting, and help
- Support for Windows conventions and subsystems, such as the Clipboard, drag-and-drop editing, and help

All of this is achieved by bringing together the worlds of object-oriented programming (OOP), graphical user interfaces (GUIs), visual design tools, and traditional business languages—all in a single, integrated desktop.

Visual Programming Tools and a Complete IDE

The integrated development environment (IDE) provides tools that enable you to *visually* design the forms, menus, reports, icons, and so on, for your applications using point-and-click and drag-and-drop techniques. These tools let you see the result of your design as it progresses.

The IDE is an intuitive and powerful environment for creating applications; for example, the Repository Explorer is patterned after the Windows Explorer which provides a consistent look and feel so you can get to work right away. The Repository Explorer allows you to browse projects, applications, libraries and DLLs, modules, entities, classes and errors.

The IDE offers a sophisticated and powerful environment for the advanced developer, with features that allow you to spend more time on the business logic aspect of application development. Visual Objects, for example, automatically tracks and maintains the relationships between the various pieces of an application for you, determining which components need to be compiled in order to build an application. Make files and compiler and linker script files are, therefore, obsolete.

Additionally, the IDE offers the capability of incremental linking when running from inside the IDE or debugging. This feature enables fast prototyping and quick feedback when you make changes to your application, it also enables you to test and debug your applications efficiently using the debugger in the IDE.

Visual Objects offers a just-in-time debugging feature. If an exception or a runtime error occurs while running an application (that has the debug option turned on in the application properties) from within the IDE, the debugger will be invoked in order to look at the error.

After developing, testing, and debugging your application, distributing it as a standalone EXE is easy. You simply click a button to generate an EXE, which can be distributed royalty-free to your end users.

A Fully Object-Oriented Language

The Visual Objects language is fully object-oriented. You may ask: Why object-orientation? There are many reasons, the most fundamental of which is that programming for event-driven, GUI environments presents a set of challenges that are aptly met by object-oriented programming (OOP).

As you read through the Visual Objects documentation (in particular, this guide and the *Programmer's Guide*), you will see how OOP naturally lends itself to GUI environments by giving you the capability to develop complex systems through standard, reusable components, in a manner that models the real world.

To facilitate object-oriented programming, Visual Objects includes extensive *class libraries* for:

- GUI programming
- Database management
- Internet client services
- Internet server applications
- Object linking and embedding (OLE)
- Reporting

These libraries provide very powerful building blocks for your applications. In addition, the visual tools in the IDE exploit the strengths of object-orientation by using these class libraries to generate object-oriented code based on your designs.

Note: Class libraries are no different from other libraries you would use in your applications—instead of containing functions, for example, they contain class and method definitions.

The language also features a structured superset of the Xbase language. (*Xbase* is the industry standard term for those programming languages that inherit from the original dBASE system, including CA-Clipper, CA-dBFast, the dBASE family of products, and FoxPro.)

The Xbase superset contains extensions for Windows and its environment, including the ability to access all Win32 Application Programming Interface (API) functions for low-level, system programming.

Open Database Access

Visual Objects gives you a wide variety of choices in terms of database access. It supports:

- Both procedural and object-oriented access to Xbase databases

  Visual Objects supports the procedural database commands and functions—such as SKIP and EOF()—that are traditional to Xbase languages.

  It also includes, however, an object-oriented interface to Xbase database management. The object-oriented interface is akin, semantically and syntactically, to the commands and functions traditionally used in procedural access. Instead of commands like APPEND, COMMIT, and ZAP, for example, you will use methods named Append(), Commit(), and Zap() to perform the same operation.

Note: With these new methods, all the capabilities of the traditional Xbase approach are provided, but have been enhanced to fit the event-driven, multi-tasking nature of GUI applications.

- Access to both Xbase and SQL databases

  When using an object-oriented approach to database management, both Xbase and SQL databases can be accessed. Furthermore, access to these two different types of databases is accomplished using a single, compatible protocol. This allows an application to manage Xbase and SQL databases with the same code.

- Several different Xbase/SQL database formats

  When accessing Xbase databases (using either a procedural or object-oriented approach), you can choose from a variety of file formats. This is accomplished through replaceable database driver (RDD) technology. With RDDs, a single application can access different database file formats using a common language interface. This allows you to tailor your applications so that migrating from one database format to another is simple and straightforward.

  Visual Objects supplies several popular RDDs, and through its open architecture allows for development of third-party RDDs. See the Replaceable Database Drivers section in the "Using DBF Files" chapter in the *Programmer's Guide* for more information about RDD technology. Refer to the "RDD Specifics" appendix in the same volume for detailed information about specific RDDs.

  Similarly, support for SQL databases is accomplished using Open Database Connectivity (ODBC), a widely used API for SQL access under Windows. This technology also uses replaceable drivers, supplied as dynamic link libraries (DLLs), which standardize the interface to the various database formats. Visual Objects comes bundled with DLLs for many of the popular ODBC formats, and provides language support for a superset of the standard ODBC API, as well as, an object-oriented interface compatible with that used for Xbase database files.

An Active Repository

Visual Objects is a repository-based system. The multi-tiered *repository* is where the IDE stores all application components, and it automatically manages the relationships between the various components of an application. If you make a change to a library component, for example, the repository automatically marks every application with that library in its search path, indicating that it should be rebuilt.

With Visual Objects you can create and use multiple repositories which are represented by projects. This allows a repository to be used on a system different from the system that created the project. Being able to use different projects makes it much more convenient to have a backup copy of your work.

A Native Code, Incremental Compiler

Visual Objects can compile your applications down to native machine code. This gives you the flexibility of using OOP without sacrificing runtime performance.

To support iterative development, the compiler works with entity-level granularity. *Entities*, as explained in greater detail later in this guide, are the smallest pieces of an application (like a function or a global variable declaration). *Entity-level granularity* means that when you make a change to an application and then build the application, the compiler determines which entities of the application have changed (or are affected by the change), and automatically recompiles only those pieces, as opposed to recompiling entire modules.

Entity-level granularity is a powerful feature because it speeds development—you spend less time waiting for your application to be built and more time designing, enhancing, and fine-tuning. It also makes prototyping fast and easy.

Visual Objects uses foreground compilation which prevents activity in the IDE while compiling your application.

A Portable Executable Format, Incremental Linker

Visual Objects uses an incremental linker to speed the development process. The linker produces EXEs in portable executable format, the new standard for 32-bit applications.

Incremental linking means that once an application has been compiled and linked to create an executable file, changes made to the application are tracked so that only modified code needs to be linked. This allows you to test and prototype applications faster than ever before.

Incremental linking of resources requires more space for an individual entity than actually needed in the EXE file, therefore, incrementally linked EXE files are bigger than non-incrementally linked EXE files. Visual Objects uses incremental linking only for temporary EXE files called DBG files. These DBG files reside in the same directory as the target EXE file and are used when running the application from within the IDE. It is important to note that certain changes to your application may require a full relink. A full relink can be forced by manually deleting the DBG files; this will force the linker to fully relink the application.

Reporting Using the Report Editor

The Visual Objects Report Editor provides powerful reporting capabilities for your applications. The Visual Objects Report Editor consists of the CA-Report Writer and the CA-Report Viewer.

The CA-Report Writer offers a sophisticated database publishing interface, allowing you to design and produce custom database reports at the press of a button. While in the CA-Report Writer, you can use its intuitive, GUI environment to define the structure and specifications of the report. For example, you can add fields, text, tables, and pictures to a report, and format the various sections (like headers, footers, and titles). The CA-Report Viewer allows you to view your reports as they are created.

In addition to creating your application reports, the Report Editor will allow you to provide your users with the ability to create their own reports from within your Visual Objects applications.

An Open Architecture

Visual Objects features extensible subsystems that facilitate the integration of third-party tools within the product. It also supports a number of powerful features that allow your applications to interact with other applications and exchange data, as well as, use routines written in other languages, such as C, C++, Pascal, and COBOL.

For example, you can:

- Access the functions stored in a DLL

  A DLL is a library of functions in which only the interface definitions are visible, not the source code. Visual Objects has the necessary language support (such as pointers and structures) to access standard Windows DLLs, including the Win32 API.

Not only can you use DLLs in your applications, but you can *build* and *debug* them in Visual Objects. Visual Objects-generated DLLs can be used with either Visual Objects applications or with foreign applications. When using a Visual Objects DLL with a foreign host, it is the user's responsibility to ensure that the exported DLL interface is compatible with the host's capabilities.

- Use Dynamic Data Exchange (DDE) to exchange information with other DDE-compatible applications.
- Interface with your system's Clipboard facility to transfer different types of data between applications.
- Use object linking and embedding (OLE) to incorporate controls, objects, and even entire OLE-compliant applications into your Visual Objects applications.
- With just a few mouse clicks you can easily enhance the power of your applications by using OLE 2.0. OLE 2.0 is a powerful way to integrate professional full-fledged applications for a wide range of areas. OLE objects can also be stored in DBF files.
- Use the Automation Server to create a Visual Objects class from the OLE Automation objects provided by third-party applications. This allows you to control applications remotely through their macro language using Visual Objects syntax. You can also create OLE Automation Server applications, ActiveX controls, and Active Server Page (ASP) components.

Now that you have had a brief overview of Visual Objects, let's learn about its new features.

Visual Objects Features

Visual Objects provides support for:

- Internet client services

Protocol implementation for low-level Internet access, file transfers, and e-mail.

- Internet server applications

  Internet interfaces for CGI applications and ISAPI libraries.

- OLE Automation server applications

  Inter-application communication via OLE automation server applications or via Web servers supporting ASP components.

- ActiveX control creation

  Inter-application communication utilizing Visual Objects windows as ActiveX controls within other applications.

- Win32 native console applications

  Alternative to Terminal Lite applications for character-based debugging and logging output.

Other major features include the following:

- Version control

  An interface for source code control provides integration with Microsoft Common Source Code Control API.

- Compiler enhancements
- Thread-safe kernel runtime
- Reindexing projects without shutting down
- Support for DLL debugging
- New system-wide options for the Repository Explorer, Source Code Editor, and Debugger
- Predefined application frameworks for the above-mentioned Internet, OLE server, and console application types, as well as additional application options
- Keyboard access for properties
- New Window Editor options, including a window type, OLEDataWindow, and new controls
- Data-aware text controls

- Interactive tab order setting
- Band-style toolbars in applications

All of this technology greatly enhances the power, flexibility, and ease that Visual Objects offers application developers of all levels and backgrounds.

As you may have surmised by now, much of the power of Visual Objects comes from its class libraries, which provide an elegant and extensible way of using supporting services. They are tightly integrated with both the object-oriented programming language and the visual design tools in the IDE. Naturally, libraries and classes have been added for Internet resources and to support OLE server components, and console applications.

Similarly, the IDE has been updated with features that reflect Visual Objects innovative technology. These include the Application Gallery, with its many predefined application frameworks and samples to show their uses.

In This Guide

This *Getting Started* guide is your introduction to Visual Objects. It contains all the information you need to get a quick and productive start, and is organized into the following chapters:

Chapter 1, Introduction, provides an overview of Visual Objects and also details the conventions and symbols used in presenting the information in this guide. Because they are vital to your understanding of this guide, it is highly recommended that you take the time to familiarize yourself with them.

Chapter 2, Installing and Starting Visual Objects, provides the information you need to install and start Visual Objects.

Chapter 3, Object-Oriented Programming Concepts, describes the basic principles of object-oriented programming (OOP).

Chapter 4, An Overview of the IDE, presents an overview of some of the features of the IDE.

Chapter 5, Learning the Basics, provides a hands-on tutorial that you can work through to build a sample Visual Objects application.

What You Need to Know

In addition to an understanding of basic programming concepts, this guide assumes that you are familiar with Microsoft Windows terminology and navigational techniques, including how to work with standard Windows items like menus, dialog boxes, the Clipboard, and the Control Panel. If you are unfamiliar with Windows, please refer to your Windows documentation before using Visual Objects.

Note: In general, when this guide indicates a procedure using toolbar buttons or mouse actions, it takes for granted that you know the alternative procedure, using only the keyboard. For example, you will be directed in most cases to "click the Find toolbar button," rather than "select the Edit Find command, press Alt+F3, or press Alt+E, F."

General Typographic Conventions

This guide also employs several typographic conventions (such as capitalization or italic formatting) to distinguish between language elements and discussion of them.

Key Names

The names of keys, such as Enter, Ctrl, and Del, appear in the document as they do on your keyboard, where possible.

Note that when referring to the four arrow keys as a group, they are referred to as Direction keys; however, the name of each Direction key (for example, Up arrow or Left arrow) is used when referring to them individually.

Key Combinations

Whenever two keys are joined together with a plus (+) sign (for example, Ctrl+R), you should hold down the first key while pressing the second key to complete the command. Release the second key first.

Key Sequences

When keys are separated by a comma (,), press them in the sequence indicated. The keystroke sequence Alt+E, C, for example, indicates that you should hold the Alt key down while pressing the E key, release them both, and then press and release the C key.

User Input Examples

The following conventions are used for user input:

- Literal information (text that the user must enter exactly as shown) is shown in bold:

  Insert the diskette into drive A and type **a:\install**.

- Placeholder text (variable information a user must enter) is denoted by a bold and italic typeface:

  Enter **login** ***username***.

UPPERCASE

The following appear in uppercase:

- Commands (like CLEAR MEMORY)
- Keywords (for example, AS, WORD, and INT)
- Reserved words (for example, NIL, TRUE, and FALSE)
- Constants (for example, NULL_STRING and MAX_ALLOC)

Mixed Case / Initial Capitalization

The following are displayed using mixed case:

- Function, method, and procedure names (like SetDoubleClickTime() and Abs())
- Class names (for example, TopAppWindow)
- Variable names (for example, oTopAppWindow and nLoopCounter)

Italic

Variable names are displayed in italic in syntax (for example, Abs(<*nValue*>)) and when referring to them in the discussion text.

Cross References

The following conventions are used:

- Guide name in italic:

  See the *IDE User Guide.*

- Part name in single quotes:

  See 'Database Programming' in the *Programmer's Guide.*

- Chapter name in double quotes:

  See "Creating an Application" in the *IDE User Guide*.

- Section name as it appears in the document:

  Also see the Saving a Program section.

Getting Help

Visual Objects provides an online help system, which can be used to display information on your console as you work. You can use any of the following Help menu commands:

Menu Command	Description
Index	Displays an index of available help topics about the Visual Objects language and IDE.
Context Help	Allows you to get context-sensitive help for an item or area currently displayed on your screen.
How to Use Help	Describes how to use the Windows Online help system.

In the IDE you can also receive context-sensitive help for a menu or menu command by pressing either the F1 key or the Shift+F1 key combination. Press Shift+F1 to receive context-sensitive help for most dialog boxes and windows.

Additionally, when the Source Code Editor is open, you can receive context-sensitive help for the keywords, commands, classes, and functions in a selected module or entity. Simply highlight the keyword, command, class, or function and press the Shift+F1 key combination.

Chapter 2

Installing and Starting Visual Objects

This chapter discusses requirements and procedures for installing and starting Visual Objects.

Installing Visual Objects

AutoStart Installation

After Windows has started, place the Visual Objects CD-ROM in the CD-ROM drive. Once this is done, the installer is automatically invoked. Follow the steps provided below under the Manual Installation instructions, skipping steps 1–3.

Manual Installation

To install Visual Objects on your hard drive:

1. Insert the Visual Objects CD-ROM in the CD-ROM drive.
2. Click the Start button, then click Run.
3. In the Run dialog 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 to invoke the Installer.

   The first window that is presented allows you to select the language that the installer will use. NOTE that this is only applicable during the installation process and not during the normal use of Visual Objects.

5. Having selected your preferred language, press the **Next >** button. The window will now display the *License Agreement*, which you should read and then agree to by pressing the **Yes** button.

6. The next window displays important information regarding the changes that have taken place since the last version was released. After reading this information, press the **Next >** button to proceed to the *Registration Screen*.

7. The registration information in order to proceed with the installation. Once you have entered this information, press the **Next>** button. **NOTE:** the *Product ID#* can be found on the back of the CD case.

8. The next window allows you to select the drive and directory for the installation. This can be achieved by using the mouse in the selection boxes, by typing directly into the path box or by using a mixture of both. If the directory that you require is not present on the disk, the installer will create it for you but you will need to type it in on the screen. Once you are happy that the drive and path are correct, press the **Next >** button to move to the *Program Folders* window. NOTE: For the rest of this guide we will assume that you have used **C:\CAVO27** as your installation path.

9. The Program Folders are displayed from the **Start/Programs** menu and help you group together items that are associated with each other. In this window you are able to select an existing folder or enter a new folder that will be created fore you. Once you are happy with the name of this group, press the **Next >** button to move to the last window. **NOTE:** The default that you are provided with is **Visual Objects 2.7** and for the rest of this guide we will assume that this has not been changed.

10. You have now provided enough information for the installation to start and this is the last chance to change things before the files are copied to disk. If you want to make change you can use the **< Back** button to move to previous screens and then come back through to this window. When you are happy with all of the settings, press the **Next >** button to start the installation.
11. The files will now be copied to disk and you will see two progress bars indicating the status of this operation. The top bar shows the current process while the bottom bar indicates the percentage of the overall operation completed along with an estimation of the remaining time required. When this operation is completed the window will be replaced with the *Setup Finished* screen.
12. The setup is now complete and the install process informs you of this by presenting you with this screen. Acknowledge the completion by pressing the **Finish** button and Visual Objects is installed.

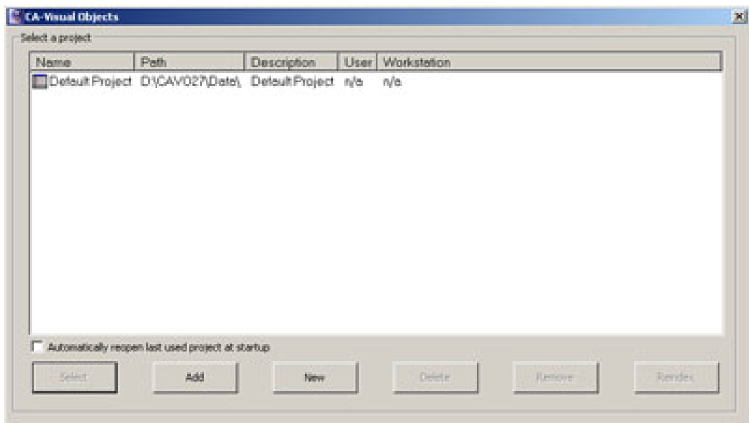
Before Starting Visual Objects

Depending on your individual requirements, you may need to adjust certain settings before starting Visual Objects. For example, you should add **C:\CAVO27\Bin** manually to your PATH environment variable. This ensures that your applications run properly from within the IDE.

Starting Visual Objects

You can start Visual Objects by choosing the Visual Objects 2.7 folder and then choosing the Visual Objects 2.7 menu item.

The Visual Objects shell window appears an inside it is the *Project Selection* dialog.



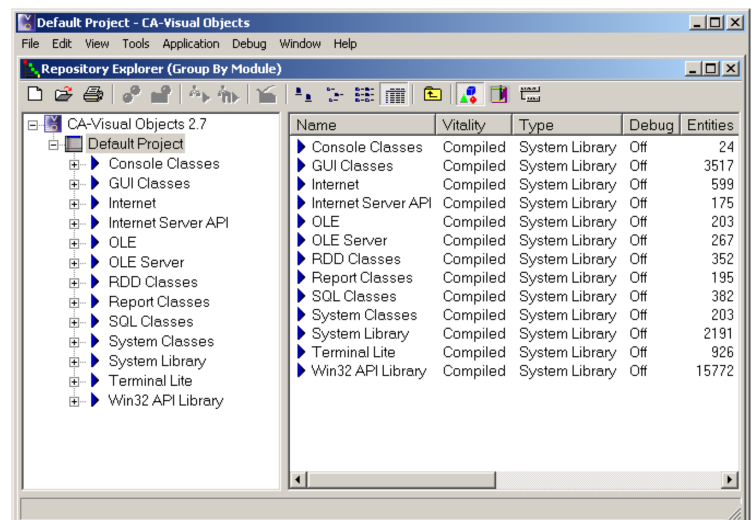
Later on you will read about the structure of Visual Objects storage and find that you can have multiple *Projects*, each of which has its own repository. Inside each repository are the applications, libraries and DLLs that make up your systems.

Visual Objects can only address one project at a time and closes the current project before opening the next. To be able to look at two or more projects at the same time, you are able to start more than one instance of Visual Objects and open a different project in each of them.

When you first start Visual Objects there is only the *Default Project* available but others will appear in this list as you create them.

For now, click on the *Default Project* entry in the list and press the **Select** button.

The IDE will now load for this project. This IDE desktop is also referred to as the *Repository Explorer*:



The Repository Explorer is the primary workspace in Visual Objects. It contains all the applications, libraries, and DLLs currently stored in the repository. Using the Repository Explorer, you can view the contents grouped by module, entity type, or class.

What's Next

Now that you have installed Visual Objects and learned how to start it, the next chapter introduces you to some of the basic tenets of object-oriented programming, which is the very heart of the Visual Objects architecture.

Chapter 3

Object-Oriented Programming Concepts

Component
Refers to both the compile-time class definition and the runtime object of that class. May also refer to a class library.

In the first chapter of this guide, it was stated that object-oriented programming (OOP) "naturally lends itself to GUI environments by giving you the capability to develop complex systems through standard, reusable *components*, in a manner that models the real world." In this chapter, we'll explore some of the basic OOP concepts and, in doing so, explain exactly what is meant by this statement.

Why Object-Orientation?

What is the motivation for learning OOP? Furthermore, what possible motivation is there to rewrite existing programs in an object-oriented fashion?

The reasons are simple. First, it's a logical adaptation of the way we already view the world. Our perception of the world is a collection of objects that interact with each other; therefore, it is natural for us to think of software development in the same way. Secondly, it's a smart business decision—objects are intrinsically modular and therefore encourage both reusability and safe, incremental enhancements. Finally, object-orientation fits the event-driven nature of GUI programming rather well. We'll examine all these reasons in greater detail in this chapter, but first, let's explore this last point.

The Paradigm Shift

Paradigm
Model of behavior.

People often speak of a "paradigm shift" when programming for GUI environments. This is because Windows applications behave differently from traditional DOS applications.

In applications developed before GUIs became commonplace, the program dominated the conversation with the user. The program asked the user for input and displayed output in return. When filling in an insurance claim form, for example, the program led the user through each item that it required.

In well-designed GUI applications, however, the user is in control. Not only can users flip between applications at will, but they can also choose what to do next in an application (for example, which fields in the form to fill in and which to leave to the imagination of the application). Windows applications, therefore, are said to be *event-driven* because the events generated by the user dictate what happens in the application. This is contrary to DOS applications in which the program has control. (For more information about moving from character mode to Windows, see the *Programmer's Guide*.)

Event-Driven Programming

Not surprisingly, the new paradigm has profound effects on how you program. Flexible control for the user means changing the way you program.

It is this shift to "the user in control" that caused developers to select object-orientation as *the* way to tackle GUI development.

Developers came to this conclusion because event-driven behavior requires very modular programming: the program must execute in tiny atomic units that can start up at any time, do their task quickly, and finish. As we previously stated and as you will soon see, objects are intrinsically modular. Object-oriented programming, therefore, naturally lends itself to GUI environments; traditional procedural programming techniques do not.

Note: This does not mean, however, that you need to throw out all your code. Procedural programming does have its place in OOP; however, instead of directing the application, it is used in smaller units to handle the different actions that the user can perform.

Thinking in an Object-Oriented Way

Thinking in object-oriented terms also involves another shift: changing the way you view software development. While object-orientation helps meet the challenges of programming for event-driven GUI environments, it doesn't mix well with conventional programming training.

Traditionally, programmers are taught to tackle a programming problem by breaking it down into the operations that need to be performed. We were taught to think about the steps involved in solving a problem. We aligned ourselves with the *processes* involved.

Solving a problem using an object-oriented approach, on the other hand, means thinking about the *things* in the system. By breaking down large, complex things into smaller, simpler components, we reach the same goal—software that solves a problem. For example, rather than seeing an inventory system in terms of reducing stock, repricing, or posting to the general ledger, we would look at it in terms of its elements—such as parts, pick orders, and warehouse bins—and their individual properties and associated actions.

It's easy to overcomplicate the difference between object-oriented programming and traditional process-oriented programming. However, in reality, it's just a matter of perspective. The plan and overall objective are the same in both disciplines—we're still creating a solution to a problem by breaking that problem into several simpler ones. The only difference is in the approach we take solving the problem.

The biggest hurdle is actually learning to think about programming differently. Once you've mastered it, though, you're likely to find OOP a more intuitive way of doing things, because it more closely resembles your natural thought processes—you already think in an object-oriented way. (And OOP introduces a world of advantages you haven't even seen yet!)

What Is an Object?

You are an object, so is the chair in which you're sitting, also this manual you're reading. In fact, as you look around, you'll notice plenty of objects. Your computer, your office, your coffee mug, your chair. You already think in an object-oriented way.

What do you notice about objects? In most cases, you'll probably notice two things:

1. The object has certain *properties* that help you figure out what it is and classify it.

   For example, a typical office chair has wheels, a seat, and a back.

2. The object has *actions* associated with it.

   For example, the chair described above can roll across the floor.

In the object-oriented world of software development, this is also true. An object—like a window or a check box—has properties (like a border or a caption) and actions (like displaying itself on the screen, in the case of a window, or toggling the check mark indicator on and off, in the case of a check box). Instead of actions, however, they are referred to as *methods* (more on this later).

What Is a Class?

To begin to think of programming in terms of the things in the system (rather than the processes), it is imperative that you understand what it means to classify something.

When we classify a thing, we create an *abstraction* that describes it. For example, the office chair that we described earlier is a thing that has wheels, a seat, and a back, and can roll across the floor. This is, of course, an abstract definition. We don't know what color the chair is, what material it is made of, and so on, but it gives us some guidelines in determining what is a chair and what isn't.

Class = Blueprint

In OOP, a *class* is an abstract definition of something. Classes are useful because they help us categorize and group things. They also make it easier to create *new* things. It's easy to create a thing if you know—according to its definition—exactly what is needed to create it. In many respects, then, a class is like a blueprint.

Consider a house. If you want to build a house, you don't go to the lumberyard, buy a truckload of lumber, nails, and paint, and then start building. Instead, you draw a blueprint of the house: without the blueprint, you wouldn't know how much wood to buy, what kind of nails you need, and so on. It would be impossible to construct an entire house without a plan.

The blueprint for a house gives the builder all the necessary information. It fully describes every part of the house—the placement of the windows and doors, the number and kinds of fixtures, the materials used in flooring, the pitch of the roof, and so on. It is important for the blueprint to be complete: for example, if no doors are shown in the blueprint, the resulting house won't have any either.

The blueprint is essentially the abstract definition for a house. In OOP terminology, then, the "House" class would be the abstract definition of a house thing (or object).

Instantiate
Create an object from a class.

The blueprint, however, is not a house—you can't live in the blueprint. If you wish to occupy the house described in your abstract definition, you will need to create, or *instantiate*, an actual house. When you instantiate a class, you get an object of that class—for example, instantiating the House class results in a House object.

Class Versus Object

Instantiation is important because a class is not very useful by itself. It can't do anything; it merely specifies the characteristics that an object of a particular class would possess and how the object would behave if it existed.

The difference between objects and classes is critical: objects exist in space and time, whereas a class is an abstract definition, a plan you use to construct those objects. (In software terminology, a class exists at compile time, whereas an object does not exist until runtime.)

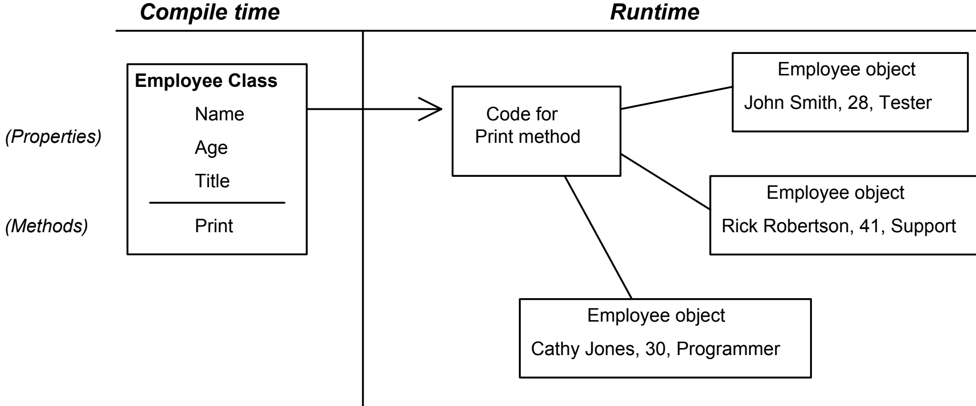
Applying Object-Oriented Thinking to Software

Let's apply what we've just learned to software. Suppose, for example, you are designing an information system for a small business to track employees, customers, inventory, sales transactions, financial records, and so on.

In this system, there are certain things that all employees have in common—for example, they all have a name, age, and title. In addition, the system should be able to print out these personal details for each employee. By setting up these basic requirements, we've just described the Employee class.

With this class, you can create a whole set of Employee objects, one for each person in the company. Each Employee has its own data (for example, John Smith, Cathy Jones, and Rick Robertson) and the ability to print.

From the one class definition, you can create multiple objects. This relationship is summarized in the following diagram:



Note: Since many objects of the same class can exist simultaneously, you can assign them to variables to uniquely identify one object from another. This is how you distinguish, for example, which Employee object you want to print or which one's name you want to know.

Methods

As you can see in the previous diagram, a class definition sets up two things for its objects: the properties it can have and the actions it can perform.

The code portion of an object—the actions that it can perform—is defined by methods of its class. In the above example, the Employee class has just one method, named Print.

Methods define what a class of objects is capable of doing. They are a lot like functions (they have parameters, declarations, programming statements, and return values), but they're different in that they are defined for a specific class and invoked for a specific object of that class.

Properties

The code inside the class (that of its methods) can see the data of the object that it is acting upon. Code outside the class (often called *external* code) usually sees only methods. This ability of the class to hide its data (or *instance variables*) from external code is called *encapsulation*.

So, how does external code get to the data? There are two ways: through *exported instance variables* or through special methods, called *access* and *assign* methods (also called *virtual variables*). The term *property* refers to either an exported instance variable or a virtual variable (in other words, any data visible to code that is external to the class is a property of that class).

Exported instance variables are sometimes frowned upon because they violate the encapsulation principle by making the object's data directly available to external code.

Virtual variables, on the other hand, allow data to be passed back and forth between an object and external code without violating the encapsulation rule. Access methods deliver data from the inside of an object to the outside, and assign methods deliver data from the outside of an object to the inside.

State

When an object is created at runtime, you can assign values to its properties and thereby change its *state*. Thus, all objects of the same class have the same properties, but the state of one object may be different from that of another. For example, all employees have a name, but one may be "Cathy Jones," while another is "Rick Robertson."

Similarly, all objects of the same class share exactly the same behavior via the methods defined in the class at compile time. The methods, however, do not change from one object to another (for example, the code for printing all objects in a class is identical, regardless of the object's state).

(The principle of encapsulation is discussed further in the Additional Strengths of OOP section of this chapter.)

Inheritance: Superclasses and Subclasses

Not only are classes useful for creating many instances of the same type of object, but they are also helpful in setting up a *hierarchy* of related classes. In this hierarchy, there is an *inheritance* relationship among the various levels—each new level in the hierarchy "inherits from" the previous, higher level.

House Example Continued

To understand this concept in an abstract sense, let's return to the house example for a moment. Imagine the architect who will design the blueprints for all the houses in a development. The houses, aside from small differences, are basically identical. Should the architect draw a set of blueprints for each house from scratch? No, that would be reinventing the wheel unnecessarily—all that is really needed is a single, generic blueprint which contains *only the details that will be the same in all houses*. From the one "master" blueprint, the architect can then design new blueprints to add the details that are unique for each house.

For example, some of the houses are to have a two-car garage, others a one-car garage. The architect, then, would design three blueprints: one master blueprint and two secondary blueprints, both of which inherit from the master but add their own unique details (one for a house with a one-car garage and another for a house with a two-car garage).

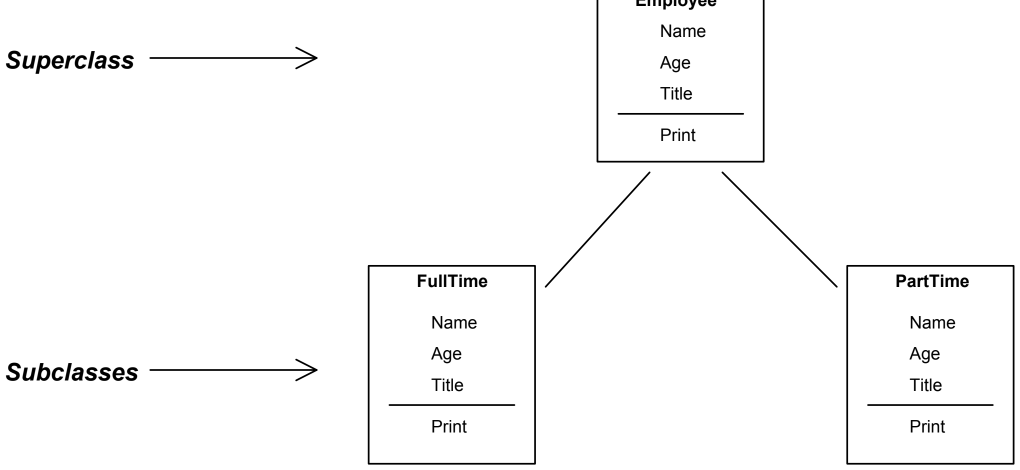
Think of the master blueprint as a starting point, and imagine that it is drawn on a transparency. When new features need to be added to the base design, the architect simply places another transparency on top of the original and draws the additions on it.

When the two transparencies are separated, the first will have only the generic design. This is the *superclass*, or parent. When the two transparencies are together, a house with embellishments is the one designed. This is the *subclass*, or child. It inherits all the characteristics of the parent but defines a more specific kind of house by adding characteristics of its own.

Subclassing the Employee Class

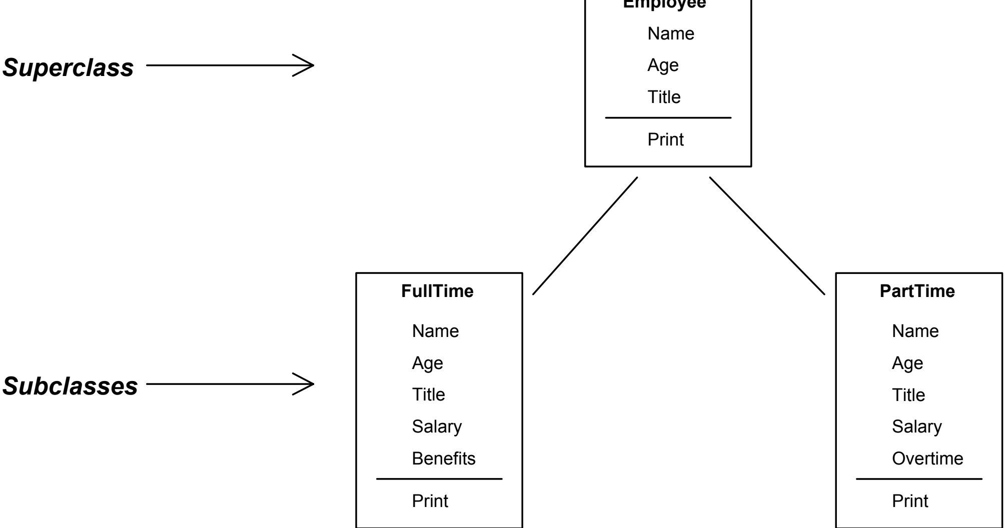
Moving back to our Employee class, suppose you decided to add a property to store the person's salary. If the person is full-time, they are salaried and entitled to benefits; if they are part-time, they are paid hourly, overtime may need to be calculated, and benefits are not granted. Therefore, it's not really just a simple matter of adding a Salary property to the Employee class—there are far too many other issues involved.

The best solution, then, is to subclass the Employee class to define two new classes, named "FullTime" and "PartTime." The new *subclasses* (FullTime and PartTime) inherit everything from their *superclass* (Employee)—both their data (name, age, title) and their methods (Print):



The subclass, then, can define its own unique data and methods, as well as modify the behavior of its inherited methods (as opposed to rewriting the originals).

For example, you need to have separate code to handle the salary calculations in the FullTime and PartTime subclasses because they are not computed in the same way. Thus, objects created from the FullTime and PartTime classes would still have all the basic characteristics of an Employee, but each would have a specialized Salary property, computed differently depending on the object's class. Each would also have properties unique to it, as shown below:



Note: The interface to these two subclasses is identical. In both cases, you refer to Salary without knowing or caring how the underlying class computes the value.

Inheritance is not necessarily only one level deep—you could potentially go on to create more subclasses and perhaps even subclasses from those subclasses. If you look at inheritance like a tree, a subclass inherits not only from its immediate parent but from *all* of its ancestors.

Inheritance, then, is the programming technique by which you adapt the behavior of a component without changing that component. Because you can give the component new behavior without destabilizing it in any way, you achieve safe, incremental enhancement. You progress from a stable status to an improved stable status.

A Real-World Example

Let's return again to the information system for which we designed the Employee class. What components would you likely need for this system? Well, a database for sure. In fact, you'll probably need several tables to keep track of employees, customers, inventory, sales transactions, financial records, and so on.

You might also want a calendar, which will be used to track shipments and billing dates. Finally, you'll probably want some mechanism through which you can generate reports (for example, total sales per month or a customer mailing list).

We're already on the path to creating an object-oriented information system, merely by the fact that we're thinking of the components of the system as classes—tables, a calendar, and a report generator.

The Table

Let's consider a single table first. We need to build the structure of the table by specifying what fields to include, what their data type should be, and other details. Next, we need to implement actions (methods) to make this table operational.

What are the methods we want to apply to a table object? Well, certainly, we'll want to add records and delete records. We'll also want to search for records, and possibly edit them. Thus, we've already determined four basic methods that will be available to every table we create.

Let's pause and take a look at what we've done. We designed a table structure, as well as, several methods that we can use to provide access to the information in that table. By doing so, we've created our first class definition. Using this abstract definition of a table as the "master blueprint," we can now continue by creating more specific table designs, adding fields and functionality specific to each table, if necessary.

> **Tip:** In fact, a class that has these methods is supplied with Visual Objects. It is called DBServer and is located in the RDD Classes library.

The Calendar

Once we've designed the table, we can move on to the calendar. To keep the design simple, the calendar will hold information by month and day only. We can represent the calendar as an array of month objects. Each month object has an array of day objects, and we schedule appointments for these days. Each day has an array of appointments.

Now that we have the calendar object's structure defined, what methods will we possibly need? We'll definitely need a method to make an appointment and one to cancel an appointment. Perhaps a method to mark employees' vacation days would be helpful, too.

The Report Generator

You've probably got the idea by now. When we go to create our report generator class, we'll have to create its structure, which might include the page format and the information source. It would have at least a single method to send output to the printer.

Communicating Between Objects

You may have noticed that the three classes we've imagined have methods that apply only to themselves. For instance, the methods that add, delete, edit, and search records in the table act only upon the table itself, not on the calendar or the report generator.

We could also, just as easily, have given our objects methods that allow them to interact with each other. For example, the table might have methods to send the name of a person who has a scheduled appointment to the calendar. The calendar would then have a method to receive that name. The calendar might need a method to send information to the report generator to print a daily schedule of appointments. The report generator might need to access information from the table to create a report.

These methods, however, would break the encapsulation of the individual classes. The right way to handle this is for a controlling object, such as a window, to manage traffic between the various objects.

Additional Strengths of OOP

We mentioned earlier that there were other benefits to OOP. Let's discuss some of these now.

Encapsulation

Encapsulation refers to the protection of the inside of something from changes made on the outside and vice versa. It is the hiding or protecting of data.

Client
The code that uses a class or the person who writes that code.

When you develop classes, it is not necessary for the *client* of those classes to understand the inner workings of the class. For example, how exactly the Employee class goes about implementing its Print method is irrelevant to the client—only the fact that it exists and works is important. To use a more simplistic example, the average person probably knows little about how a television works, and just cares that it turns on and off, switches channels on command, and presents a high-quality picture.

From a software perspective, encapsulation plays an important role. When you design a class, you can hide certain data so that when an object is instantiated from the class, the data is invisible to other objects. This allows *controlled access* to data: the only way that code outside the class can touch protected data is through methods or properties. Any external modification to the data, therefore, is done only with "permission."

If we consider the television example again, a person really should not bypass the volume control and manipulate the volume by touching the television's internal working parts. Encapsulation, in the form of the television case, prevents this. By hiding internal details of objects, and giving access only to things that should be accessed, encapsulation provides a simple and safe framework for working with objects and the data they hold.

Modularity and Reusability

Standard Components = Reusability

When constructing a house, you always use prefabricated components: girders, door frames, sink units. When constructing a computer, you assemble prefabricated integrated circuits, power supplies, and disk drives. Nobody would consider producing such complex components from concrete and raw timber, or from silicon and iron ore.

The same should be true for software—yet, when constructing software applications, often equally complex, the tradition of using prefabricated components is not as well established. The proliferation of OOP, however, has started to address this shortcoming.

Object-orientation extends the developers' ability to write modular, reusable code. Objects are essentially packaged code and data. Bolstered by encapsulation and inheritance, objects become powerful application building blocks. Future applications will be easier to create because they will be programmed by simply assembling component parts—imagine constructing a personnel package by just bundling together the payroll system and employee benefits components!

Modularity = Safe, Incremental Enhancements

In the long run, however, software construction is probably going to be more demanding than the construction of a house or computer, because it continuously requires adjustment and elaboration.

For this reason, the standard components must be easy to modify and adapt to new uses and circumstances. The architecture must allow the continuous rearrangement of components and addition of new components.

The tenets of object-orientation hold that proven code rarely needs to be touched when enhancements or other changes are necessary. Unless code written in the past is found to be incorrect—or incompatible with future interface designs—code need not be modified. Instead, changes are made by creating a subclass through inheritance and coding only what is new or different. Only one version of any piece of code, therefore, need ever exist—code is reusable.

In addition, class definitions can be grouped together in user-defined libraries. Over time, these libraries can grow to form powerful application building blocks. Since new functionality is added via inheritance, source code in the class library never changes (and consequently, user-defined class libraries are usually easier to maintain than function libraries).

Because objects are intrinsically modular and reusable, programmers automatically achieve several benefits:

- There is less code to write and debug. The development cycle, therefore, is streamlined and more productive.
- The quality of resulting applications is higher, since reused components are more frequently used and therefore better tested.
- Reusability decreases maintenance—it is easier to make changes (both enhancements and corrections) without side effects.
- Because code is of higher quality and is better tested, and because development time is streamlined, programmers can focus more on design, creating applications that are more sophisticated and robust.

Summary

Hopefully, it's becoming clear that object-orientation is a good way to manage GUI applications. OOP is the answer to many of the programming complexities and challenges presented by GUI environments. Windows development is the perfect place to put object-oriented theory into practice.

It doesn’t stop there, however, applications in general can be thought of and implemented in object-oriented terms, whether or not they are GUI, whether or not they utilize databases, or even if they perform only rudimentary tasks.

The Visual Objects Libraries

To facilitate object-oriented programming, Visual Objects includes a set of extensive libraries. These libraries provide very powerful building blocks for your applications, and offer an elegant and extensible way of using supporting services. They integrate well, not only with the programming language but also with the IDE (for example, the Repository Explorer). Libraries also provide an extremely effective way of insulating application code from platform-specific implementation details.

Visual Objects provides the following class and function libraries:

System Classes	OLE Server
RDD Classes	Report Classes
SQL Classes	System Library
Internet	Terminal Lite
Internet Server API	Console Classes
OLE	Win32 API Library

System Classes

This library defines classes that are used by the other system class libraries (that is, GUI Classes, RDD Classes, SQL Classes, and Report Classes). Whenever you associate one of these class libraries with your application, you should also associate System Classes with it. To use the code generated by the Menu, DB Server, FieldSpec, SQL Server, and Window Editors, you *must* associate this library with your application.

GUI Classes

This library contains over a hundred classes that allow you to create the objects required for a full-featured GUI. It includes facilities for creating forms, menus, pushbuttons, scroll bars, status bars, list boxes, and so on, and also includes simple shapes and other abstractions associated with a GUI.

Using the GUI Classes library also gives you access to the Standard Application (which you will explore thoroughly in "Learning the Basics" later in this guide) and other predefined application frameworks, robust error and exception handling, and a wide range of stock objects, such as useful icons, bitmaps, and colors.

You must associate this library with your applications if you plan to use code generated by the Window Editor and/or the Menu Editor.

RDD Classes

This library provides an OOP interface to Xbase DBF files using classes and methods instead of traditional commands and functions. It must be associated with your applications if you plan to use code generated by the DB Server Editor.

SQL Classes

This library provides an OOP interface to SQL tables using classes and methods instead of traditional SQL statements, and must be associated with your applications if you plan to use code generated by the SQL Editor.

As described earlier, SQL database access is accomplished using the ODBC protocol. For your convenience, therefore, this library also contains ODBC API function definitions that can be used to program directly to the ODBC API.

Note: The ODBC API functions are not specific to Visual Objects and, therefore, are not included in our documentation. Refer to the standard ODBC documentation provided by your ODBC vendor for details about these functions.

Internet

As described earlier in the Visual Objects Features section in the "Introduction," the Internet library implements support for common Internet client services, including the following protocols: File Transfer Protocol (FTP), Simple Message Transfer Protocol (SMTP), Post Office Protocol (POP), User Datagram Protocol (UDP), and Transmission Control Protocol (TCP).

Internet Server API

Additionally, the Internet Server API library is a framework for creating Internet server applications using either the Common Gateway Interface (CGI) or Microsoft's Internet Information Server API (ISAPI).

OLE

This library is an extension of the GUI Classes library and provides client support for Object Linking and Embedding 2.0. If you use this library, you also have to include the GUI Classes in the search path of your application.

OLE Server

The OLE Server library implements support for creation of OLE Automation Server applications, ActiveX controls, and Active Server Page (ASP) components.

Report Classes

This library provides an OOP interface to the CA-Report Writer and the CA-Report Viewer, and must be associated with your applications if you plan to use code generated by the Report Editor.

System Library

This library provides basic system function support. This library also provides support for traditional Xbase database functions like DBCreate() and EOF(). It is automatically associated with every Visual Objects application.

Terminal Lite

This library is a limited set of compatibility functions for traditional Xbase screen I/O techniques. You should associate this library with your applications only if you want to display your output for debugging or logging purposes.

Console Classes

As described earlier, the Console Classes library is an alternative to the Terminal Lite library for character-based debug/logging output. While the Terminal Lite library emulates character mode in a GUI window, the Console Classes library utilizes the Win32 native console application support.

Win32 API Library

This library contains Win32 API function, constant, and structure definitions. You should associate this library with your applications only if you plan to exploit low-level, system programming.

Note: The Win32 API functions are not specific to Visual Objects and, therefore, are not included in our documentation. Refer to your *Microsoft Win32 Software Development Kit* for details about these functions. If you have been using the Windows API with your CA-Visual Objects 1.0 or 2.0 applications, you might encounter certain incompatibilities in the API. For differences in the API, see the Win32 SDK.

What's Next

To gain a better understanding of the features available to you, the next chapter points out some of the various features of the Visual Objects development environment.

Chapter 4

An Overview of the IDE

This chapter presents an overview of some of the features of Visual Objects. Its purpose is to help you gain both an understanding of what features are available to you, as well as a familiarity with the basics of working in Visual Objects, so that you can go on to complete the sample application introduced in the next chapter.

Note: This chapter only touches upon some of the tools provided by Visual Objects. For complete details, please refer to the *IDE User Guide* and the online help.

Repository-Based Development

Before you start to use Visual Objects, you need to understand the implications of moving from *file-based* development systems to a *repository-based* development system.

No Need to Work with Files

First of all, you do not have to deal with files when working with Visual Objects. Instead of an application that is comprised of one or more files (PRG, CH, and so on), an application now consists of one or more *modules*. In addition, all the items that were in your files—such as functions and procedures—are now referred to as *entities*. All your applications now reside in a *project.* Each project consists of a repository that contains all your applications, DLLs, libraries, modules, and entities.

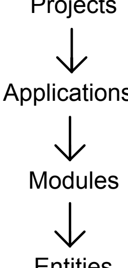
In Visual Objects, all of these things—applications, modules, and entities—are stored in a multi-tiered repository. While they are all still manageable, editable pieces of the application, they are no longer file-based—the repository holds them all. (For example, if you import source code from another application into the repository, there is seldom a need to work with external files of any kind once the files are imported.)

Note: Modules in the repository can be linked to external files if you prefer to maintain a file-based application. Visual Objects provides File Import and Export commands that you can use for maintaining backup files.

Visual Objects multi-tiered repository is broken into two parts: the System part and the User part. The System part contains all information specific to the Visual Objects system. This portion of the repository is read-only and cannot be affected by the developer. The User part can contain an unlimited amount of user projects, available to be stored either locally or on a network. The user projects inherit their attributes from the System part, creating a complete repository.

An Internal, Automated MAKE Facility

The repository manages all of the pieces of an application for you. It automatically maintains the relationships between the various entities of an application. Each time you build an application, the repository "knows" what to compile based on changes that you have made and builds the application in the most efficient way. Such automation eliminates the need for make files and compiler and linker script files.



The repository is based on a hierarchical, object-oriented model. In Visual Objects, *projects* contain *applications* (like "Order Entry") and libraries (like the GUI Classes library) that consist of *modules* (such as "Standard Shell"), which in turn consist of *entities* (such as "CLASS Standard Shell Window" and "METHOD StandardShellWindow:Init"). The highest level in the hierarchy, naturally, is the project. A project displays the applications defined in the repository. The Visual Objects repository system allows the users to create multiple repositories that can be used on different systems.

The *applications* contained within a project can be defined as one of three types: executable, library, or DLL. Specifically, an *executable* (EXE) is exportable as an executable file and a *DLL* as a dynamic shareable file, whereas a *library* is used only at compile time and is included in an EXE or DLL file. A listing of libraries included in the application can be found in the Application Options dialog box. You can also add and remove libraries from the application using this dialog box.

Modules, which form the third level in the hierarchy, are in many ways comparable to traditional source files (for example, PRG files). They contain a group of logically related parts of the application, and may be used to limit the visibility of variables, functions, classes, and so on, defined in the module.

Similarly, just as a typical PRG file contains function and procedure declarations, modules in Visual Objects contain *entities*. Entities form the fourth level in the hierarchy. An entity is any part of your application that has a name and can be edited. Some of the available entity types are:

- forms
- menus
- reports
- procedures
- functions
- resources
- classes
- methods
- structures
- globals
- constants

The IDE Tools

Visual Objects features an integrated development environment (IDE) that provides you with a flexible, intuitive, and powerful environment for creating applications.

The IDE provides a rich set of *tools* that can be used to create sophisticated GUI applications. Like a hammer or ruler, a tool allows you to create things. For example, there is a *Repository Explorer* which lets you organize and view your data, and *editors* which allow you to create forms, menus, source code, databases, reports, and icons.

The Repository Explorer

In Visual Objects, *projects* consist of *applications, applications* consist of *modules*, and *modules* consist of *entities*. When you start Visual Objects, the Repository Explorer is automatically loaded and visually represents the projects, applications, modules, and entities.

Selecting a project from the *Project Selection* dialog displays the applications defined for it, double-clicking on an application displays the modules defined for that application. Double-clicking on a module displays the entities defined for that module.

You can control the overall display of the Repository Explorer by using the Group By Module, Group By Type, and Group By Class toolbar buttons. Note that you can customize the Repository Explorer's right pane, or *list view* pane, by using the Large Icons, Small Icons, List, and Details toolbar buttons. You can also use the View Options menu command to limit the display by name and type. (See Customizing the Repository Explorer section of the "Using the Repository Explorer" chapter in the *IDE User Guide* for details.)

Visual Editors

Many of the editors in Visual Objects are *visual* and, in almost all cases, the flexibility and ease-of-use provided by the visual editors can help you work more efficiently. Their point-and-click, drag-and-drop design approach and WYSIWYG environment allow you to develop an application *visually*, thereby improving the quality of the application and reducing the total development time.

Instead of working directly in programs with the Visual Objects language, you can lay out the visual aspects of the application and much of its functionality. This provides ongoing evaluation of the application as it is created, as well as meaningful feedback about the design.

For example, to add controls (like check boxes or list boxes) to a form using the Window Editor, you simply click on an icon in a tool palette and click in the form to place it. You can then manipulate and define the control as desired (for example, resize, change colors and fonts, or add code to handle events). The Window Editor also provides a Test Mode option so that you can see what your form looks like at runtime.

Likewise, when designing a menu in the Menu Editor, it is displayed in a partially operational “preview” menu bar, so you can view what your menus look like as you create them. This preview area is continually updated as you work, providing immediate visual feedback.

Creating an application in a visual fashion improves the quality of the application and reduces the total development time, as it leads to a better definition of what is needed and thereby provides for an application that best meets the user’s needs.

Generating Code

When you are finished designing in any of these editors and have saved your work, Visual Objects generates powerful and straightforward object-oriented code based on the underlying class libraries.

For example, creating a form in the Window Editor will generate a subclass of the Window class. The generated code is not only efficient and powerful; it is clean and maintainable and forms a solid foundation for the future evolution of the application.

A Complete Development Environment

Of course, Visual Objects provides a host of other complementary tools to complete the development environment, including a source code editor, a compiler, and a debugger.

The Visual Objects IDE is designed to provide a productive framework for developing all kinds of applications—including mission-critical business systems—and is specifically designed to support the iterative development paradigm.

All development tools provided in Visual Objects are closely integrated with the repository. In fact, all aspects of working with application components—looking at them, analyzing their relationships, and editing them—are done from the repository. This ensures efficient development and protects the integrity of your applications.

The Repository Explorer

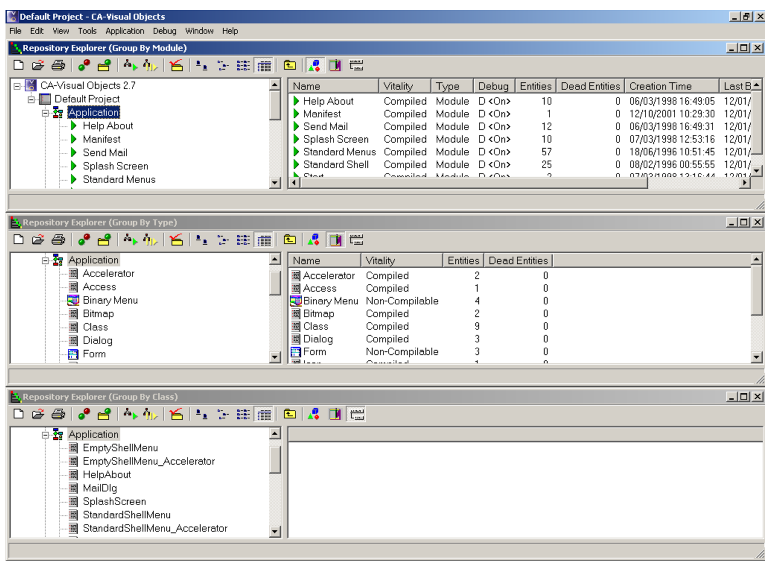
The Repository Explorer provides a convenient and organized way to view the data that is currently stored in your repository. In Visual Objects, you can view:

- Projects
- Applications, libraries, and DLLs
- Modules
- Entities
- Classes
- Errors

The Repository Explorer in Visual Objects can be customized to display a particular subset of data. For example, by clicking on the Group By Class toolbar button in the Repository Explorer classes are displayed in a collapsible/expandable tree structure that lets you determine what information to display.

You can also open multiple Repository Explorers that show different views of an application.

For example, the first Repository Explorer can show an application grouped by module, the second can show the same application grouped by type in order to display the applications entities, and the third view can show the same application grouped by class:



In addition, the close integration of the Repository Explorer with the repository provides easy access to the various editors. For example, double-clicking on a source code entity (such as a GLOBAL or a METHOD) in the Repository Explorer's list view pane loads the code for that entity in the Source Code Editor, while double-clicking on a window entity loads the form definition in the Window Editor.

The Repository Explorer, therefore, serves a variety of purposes. The views that it provides gives you an overall picture of the data that is stored in your repository. The Repository Explorer also allows you to manipulate that data—for example, you can rename an application, move a module to another application. You can even move an application to a different project by using two instances of Visual Objects looking at different projects, all with the use of the Edit menu cut/copy and paste options. Finally, it provides access to the various Visual Objects editors.

Managing Projects

Note: As discussed earlier in the introductory chapter, Visual Objects now provides version control for managing your applications. For detailed information, see Using the Source Code Control Interface in the online help.

In Visual Objects you can create and add multiple projects that access separate repositories. You may also rename or delete a project from the Repository Explorer. All the projects that are available to you can be managed through the *project catalog*. When a new project is created, it is automatically added to your catalog. You can also add a new project as long as it is not open in another instance of Visual Objects by you or another user.

You can remove a project from your catalog by pressing the Remove button. This removes the project from the Project Selection dialog, but the directory and contents of the repository are not deleted. You can, however, delete the project from the Project Selection dialog by clicking the Delete button and this will remove the directory with contents as well as taking it out of your list.

From the root level of the Repository Explorer, you can use the File menu commands, New Project, Open Project and Select Project. To remove or delete a project from your list, select the File/Select Project option to open the Project Selection dialog. From here you are able to perform the required operation but you can not remove/delete a project that you are currently working in or have open in another instance of Visual Objects.

Browsing Applications and Modules

As you have already learned, in the Visual Objects hierarchy, *projects* consist of *applications*; *applications* are comprised of *modules*, which contain *entities*. The Repository Explorer tree structure follows this top-down hierarchy.

The Repository Explorer allows you to view and use multiple projects but only one can be open in an instance of Visual Objects. For this reason, you are able to start Visual Objects multiple times with a different project in each.

The Repository Explorer allows you to view and maintain what is currently stored in the repository. The second level of the tree structure represents the project that is currently available. Under the project level is the application level. This level represents all the executables, libraries, or DLLs that are stored in your repository.

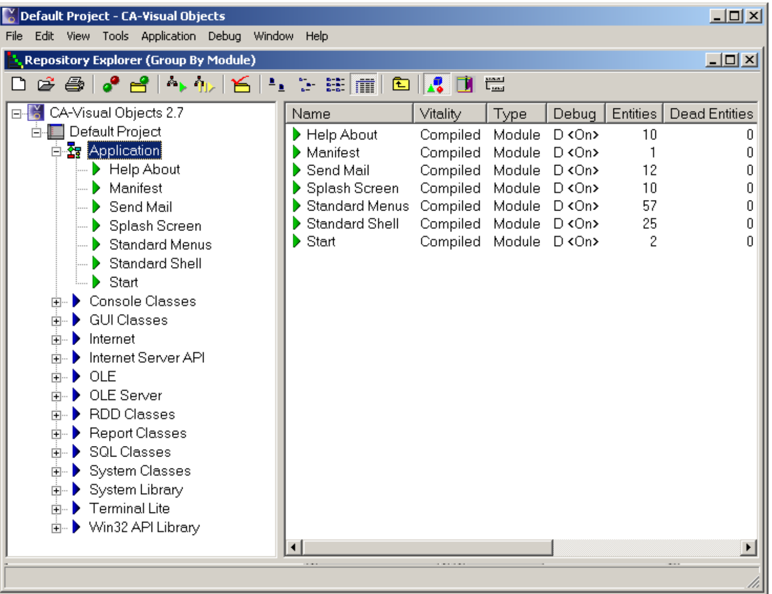
The module level is displayed under the application, library, or DLL level. All modules that are in the application are displayed here. By clicking on the individual modules, a list of all entities will be displayed in the Repository Explorer's list view pane.

Initially, the Repository Explorer displays all of the libraries supplied by Visual Objects. As you start to add to the repository, creating your own executables, libraries, and DLLs, the Repository Explorer will display them all.

You can customize the Repository Explorer as you work. You can change the Repository Explorer's initial display in several ways: specifying the size of the icons used, displaying data in list or detailed format, and restricting the display to a specified application type(s) and/or name(s). This is in addition to selecting module, type, or class view by clicking the Group By Module, Group By Type, or Group By Class toolbar button, respectively. See Customizing the Repository Explorer section in the "Using the Repository Explorer" chapter of the *IDE User Guide*.

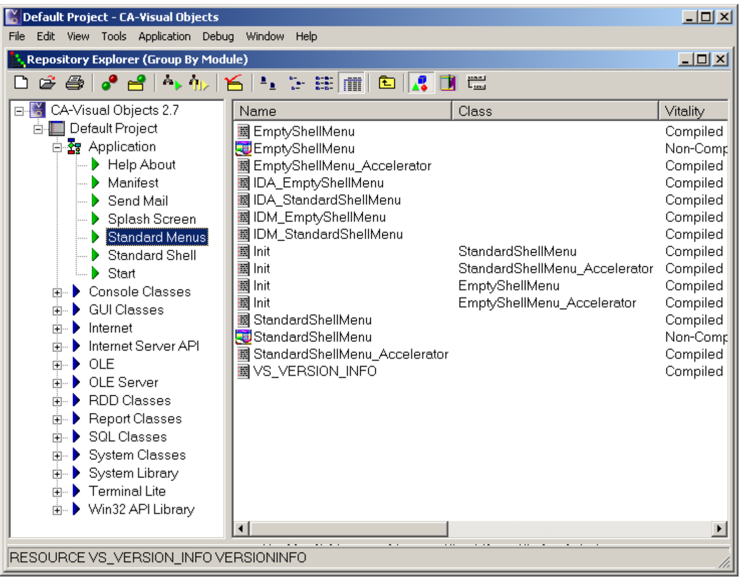
Viewing Entities at the Module Level

Visual Objects allows you to view only the entities defined for a specific module in an application. Clicking the Group By Module toolbar button displays the modules of an application. This is the default setting in the Repository Explorer. To expand this view, click on the application or click on the + icon to the left of the application. All of the modules defined for this application are displayed in the Repository Explorer tree:



Note: The Application shown is what would be displayed if a Standard MDI application were created. We will cover creating applications later in this guide.

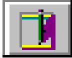
Clicking on one of the modules in your application will display its entities in the list view pane of the Repository Explorer. For example, clicking on the Standard Menus module displays the entities for this module in the right, or *list view*, pane:



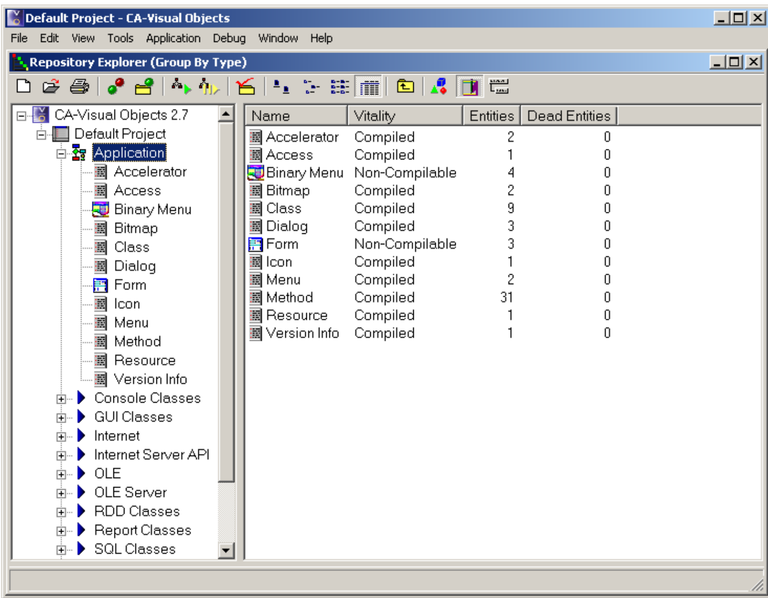
Notice that the entities are displayed sorted by entity name in alphabetical order. You can, however, sort the entities by other criteria—such as vitality or entity type—simply by clicking on the appropriate column header in the list view pane. Clicking on the column heading again will change the order to ascending/descending.

Double-clicking on a binary entity at the entity level starts an *editor*. For example, double-clicking on a form entity invokes the Window Editor, while double-clicking on a function entity activates the Source Code Editor.

Viewing Entities at the Entity Level



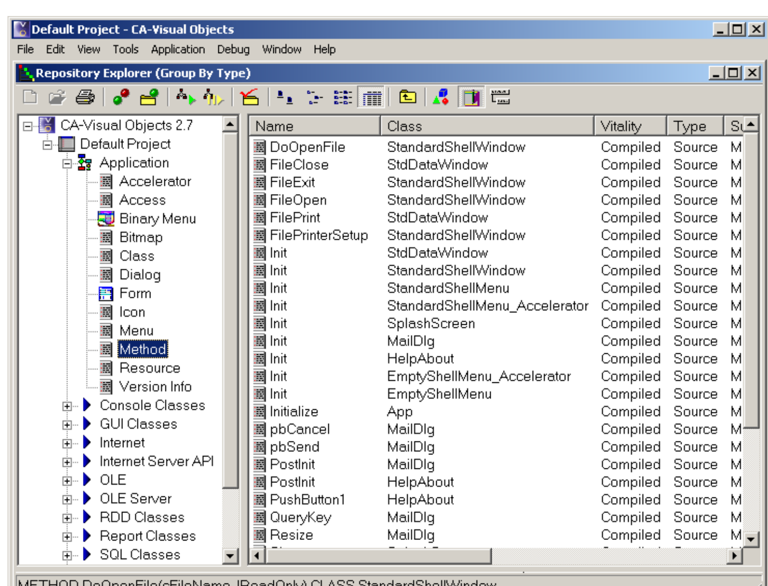
To view all entities within an application, choose the Group By Type toolbar button. For example, choosing this toolbar button for the Order Entry library would display the following:



Note that the items in this view are displayed in groups that represent the type of entity within the application.

This can be useful when you are trying an entity in a large system. For example, if you needed to find a method of the StandardShell class, you could easily click on the Method folder in the *Tree View* to display a list of all of the methods and then pick from there.

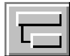
If you click on the Method module , the Repository Explorer displays all the entities and details in the list view pane:



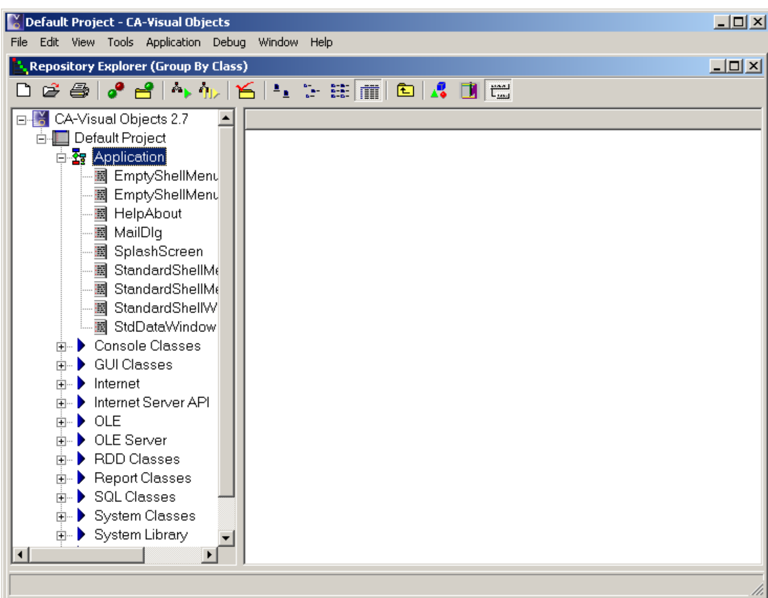
Again these entities are listed in alphabetical order and the class that they belong to is listed in the second column. A single click on the Class column will change this into displaying in alphabetical order grouped by class.

Note: You may also use the View menu commands to collapse and expand branches. See the Browsing Classes section in the “Using the Repository Explorer” chapter of the *IDE User Guide*.

Browsing Classes



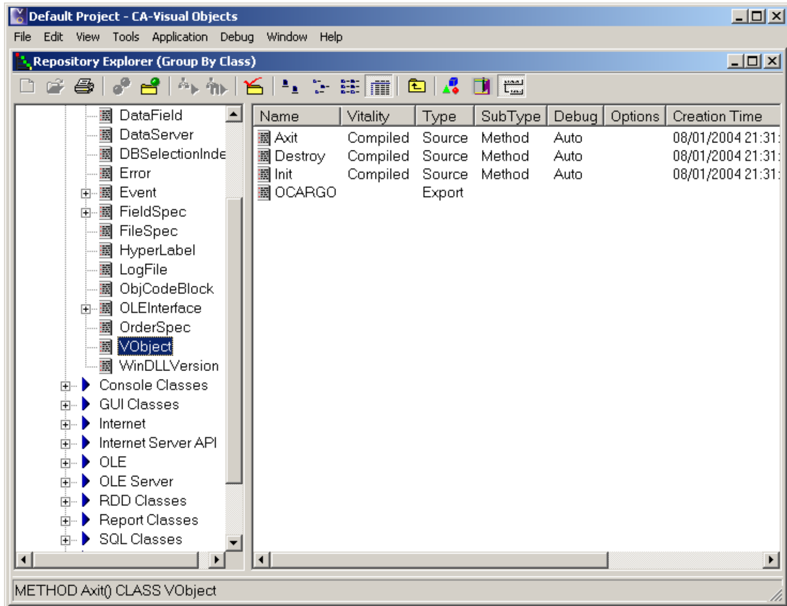
Visual Objects allows you to view all classes associated with an application. This can be done by clicking on the Group By Class toolbar button and then expanding the branch:



Notice that when grouping by class it is possible to get a single group with the title *no classes found*. This is because the classes are linked to libraries and will need to be compiled before they can be displayed as they are shown above.

While clicking on any of the groups in the Tree View will chow you the relevant code in the current application, it is also possible to extend that view to include all of the library code supplied with Visual Objects.

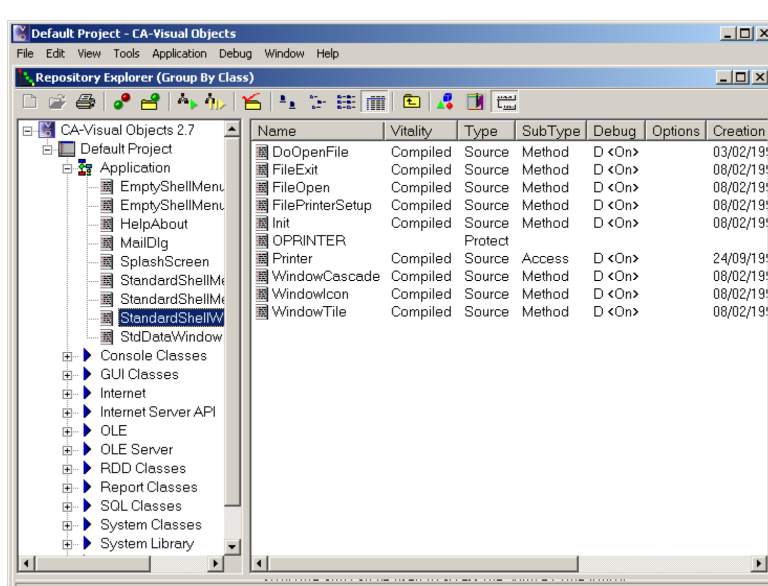
In order to extend the view of these classes, choose the View/Include Libraries menu command. The Repository Explorer now displays the classes associated with this application. Clicking on the VObject group will display details regarding the class entities:



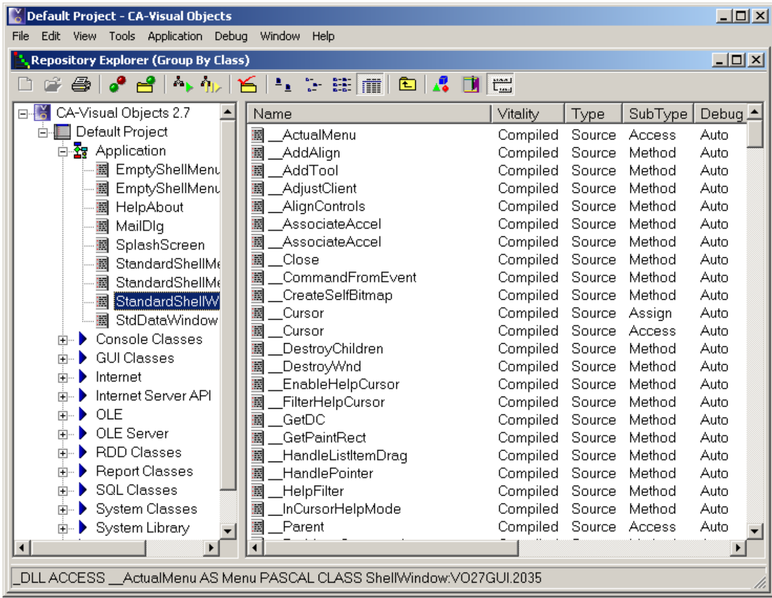
However, as you can see, this view is only showing the details of the selected class and there is very little to show in this view but, as you will find out later, the VObject class is one of the most used classes in Visual Objects. This is because almost everything inherits from this class.

In order to extend the view to be of more use, turn off the Include Libraries option by choosing View/Include Libraries from the menu.

Click on the StandardShellWindow group in the Tree View:



Now choose the View/Include Inherited menu command:



The Repository Explorer now displays the properties and methods, not only of the class but of all the classes that it inherits from as well.

NOTE: It is of special importance to note that there are many methods that start with __ (a double underscore). These methods are considered to be internal and you should not need to use them. As such you will not find them documented anywhere. They are subject to change at any time and without any guarantee of backward compatibility.

Error Browser

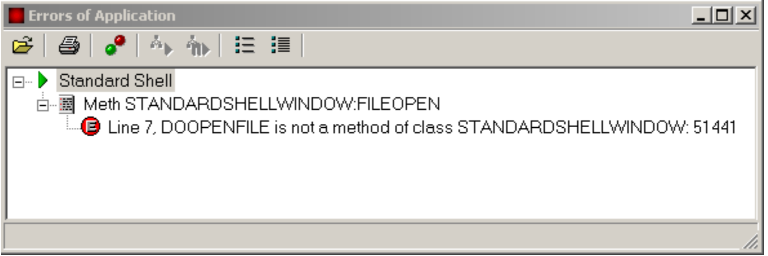
During the development cycle, compiling (or *building*) an application often results in errors. To help you locate and correct errors and warnings quickly and efficiently, Visual Objects provides an *Error Browser*.

The Vitality column of the Repository Explorer displays the compilation status of an application, module, and entity. There are three compilation states: Compiled, Uncompiled, and Non-Compilable. Compiled means that all entities were compiled with no errors; Uncompiled means that the entity needs to be compiled or has been compiled and that there are errors; and Non-Compilable means that the entity does not require compilation, (for example, any binary entity). If you have customized your Repository Explorer so that the Vitality column is not displayed, you can use the applications icons to determine the entity status:

Icon	Status
	Compiled modules
	Uncompiled modules
	Compiled entity
	Uncompiled entity
	Non-Compilable entities represent binary entities. These icons will be different for the different binary entities.

Therefore, when you build an application and a module icon is marked by a yellow dot, this lets you know that the module contains one or more entities that have errors which will be marked with a red dot.

To quickly view and go to these errors, choose the Tools Error Browser menu command. Choosing this command lists all the entities in the application that have errors or warnings. For example:



In the Error Browser, LED-style icons denote errors and warnings. An error is indicated by a red circle with an "E" inside it. Similarly, warnings and their corresponding severity levels are indicated as follows:

Icon Color	Severity Indicator	Description
Dark yellow circle	"1"	Level 1 warning (most critical)
Yellow circle	"2"	Level 2 warning
Light yellow circle	"3"	Level 3 warning
White circle	"4"	Level 4 warning (least critical)

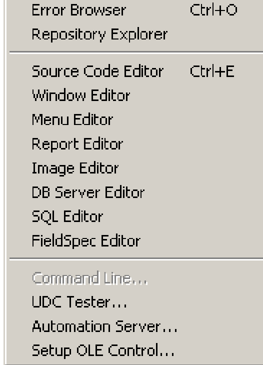
Similar to the Repository Explorer, the Error Browser displays the entities in a collapsible/expandable tree structure. If you double-click on an error, you are brought directly to the line in the source code that contains the error.

The Editors

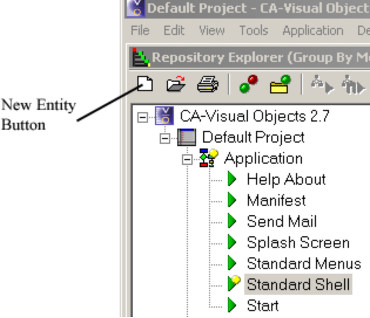
Visual Objects provides the following types of editors:

Editor	Creates
Source Code Editor	Source code entities, like functions, procedures, globals, etc.
Data Server Editors	Data server entities, as subclasses of the RDD, SQLSelect, and FieldSpec classes.
Window Editor	Window entities (like data forms and dialog boxes), as subclasses of the various Window classes.
Menu Editor	Menu entities (like menus and accelerators), as subclasses of the Menu and Accelerator classes.
Report Editor	Report entities, as subclasses of the ReportQueue class.
Image Editor	Icon, bitmap, and cursor entities (in the form of .ICO, .BMP, and .CUR files, respectively).

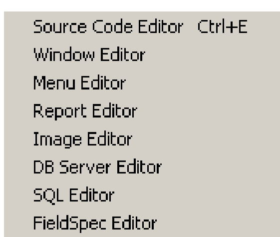
These editors can be used to create all the components of a sophisticated GUI application. All editors can be started by choosing a command from the Tools menu:



Alternatively, you can also start an editor by clicking the New Entity toolbar button in any module



and choosing an editor from a local pop-up menu:



Note: For detailed information on how to use the visual editors, see the IDE Tools section of the "Working in the Desktop" chapter of the *IDE User Guide*.

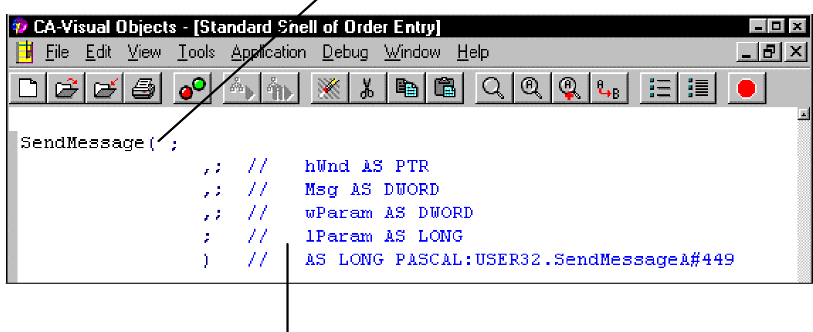
Source Code Editor

Note: For detailed information about the Source Code Editor's new features—such as using parameter tips, adding bookmarks, and displaying its search history, all discussed earlier—see the relevant topic in the online help.

The Source Code Editor provides a powerful environment for writing and editing code. For example, you can cut, copy, paste, delete, search for, and replace text, as well as undo and redo editing actions, using standard Windows techniques.

You can also fill in an incomplete function or method call with its prototype by right-clicking after the first parenthesis or by choosing the Edit Insert Prototype menu command. An Expand Prototype pop-up menu will appear; if you click on it, the prototype is inserted—this is a useful feature if you have simply forgotten the correct syntax. For example, here's the Visual Objects-inserted prototype for the SendMessage() function:

Right-click here to insert prototype



Prototype inserted after clicking on Expand Prototype pop-up menu command

If you want to view a prototype without inserting the prototype, just simply right-click on the function or method and a GOTO pop-up menu will appear. This pop-up contains the prototype and definition of an entity. If clicked on, it will bring you to the existing code in the application. For example, if you click on the StandardShellWindow class the following will display:

```
GOTO: CLASS StandardShellWindow INHERIT ShellWindow
GOTO: METHOD Init( oOwnerApp ) CLASS StandardShellWindow
```

The Source Code Editor also provides visual feedback by continually parsing each keystroke as you enter source code (or import or paste text) to color-code text based on its structure. Keywords, literals, and comments, for example, are all displayed in different colors, while each entity is separated from the next by a horizontal marker.

Additionally, the Source Code Editor provides auto indentation, source code translation from OEM to ANSI, automatic method insertion, case synchronization, and also allows you to set keyword case, tab stops, and preset breakpoints to be used in the debugger. See the Setting System-Wide Options in the "Working in the Desktop" section of the *IDE User Guide*.

The collapse/expand icons, available for every entity loaded, allow you to collapse entities that you are not currently editing to provide a cleaner view of the source code and to expand them again when you need to work with them.

Data Server Editors

One of the primary tasks of any GUI database application is to enter, modify, view, and utilize the information stored in databases. This is facilitated by the use of ancillary information, like index files in the Xbase model and WHERE and ORDER BY clauses in the SQL model.

The DB Server and SQL Editors

Visual Objects provides a set of editors—the *DB Server Editor* and the *SQL Editor*—that let you create and modify *data servers*. A data server is a high-level, abstract entity designed to give you a consistent object-oriented interface for your database. The DB Server Editor creates data servers based on the traditional Xbase model of a database file, while the SQL Editor creates data servers based on the SQL model of a table.

With both the DB Server and SQL Editors, you can import an existing database or table structure and generate a default set of field specifications (explained below in The FieldSpec Editor section) that you can optionally modify. The DB Server Editor also lets you generate a database file (and index files) from the data server definition using the File Export command.

Note: Visual Objects does not have the capability of creating SQL tables.

Using data servers offers you some significant benefits. For example, many of the properties that you define for a data server and its field specifications are designed to be used by data aware windows that you create using the Window Editor. Thus, you need only define the attributes for a data server once, and they will be automatically inherited and used by any data window that is linked to that data server.

Similarly, changes to a data server (such as the validation rules or picture formats for one or more fields) need only be made in one place, the data server itself. Resources that use the data server will automatically inherit those changes.

Using a data server also provides an integrated view of all the pieces of information related to it. Without this comprehensive entity, you would have to create and maintain the various pieces (tables, index files, relations, and field specifications) independently. Additionally, creating data servers for your database tables allows them to be easily viewed and manipulated within the IDE (for example, by using the Group By Class and Group By Type views of the Repository Explorer).

The DBServer Editor has support for the .NTX, .CDX and .MDX indexing formats and supports Character, Numeric, Logic, Date, Memo, and OLE field support.

The FieldSpec Editor

In many cases, the different data servers your application uses contain similar, if not identical, fields (for example, all zip code fields are typically the same, regardless of where they are used). You can either define the properties of these common fields (such as validation and formatting rules) each time you create a new data server, or you can create a single field specification and reuse it in each data server that needs it.

A *field specification* created in the FieldSpec Editor is essentially a set of properties that are related to a field but are *independent* of any particular data server. Thus, multiple data servers can access the same property values for common fields. If you create a Salary field specification, you can simply reuse its properties when creating an EmpSalary field in a data server for an Employee database. Additionally, if you change a field specification, the change will automatically propagate to all appropriate places.

Window Editor

The Window Editor is used for the interactive design of the various windows of your application. These windows are generated by created forms in the Window Editor. When working in the Window Editor, the windows are referred to as forms and a binary form entity is created in the Repository Explorer.

Window Types

You can create several types of windows in the Window Editor, based on subclasses of the standard GUI Classes Window class. For example, you can create MDI shell windows, *data windows*, *data dialogs*, and dialog boxes.

You can also create an OLE data window which was added to assist in the support for the OLE 2.0 standard. For more detailed information, refer to the online help system.

Tool Palette

To design these windows, the Window Editor features a floating tool palette. To place a control on a form (such as a push button, list box, or scroll bar), just click a button in the tool palette and click in the form.

You can then go on to define *properties* for your forms and the various controls you place on them (for example, you may want to specify the text that should appear in the status bar when a form or control is selected, or an ID for use in a context-sensitive help system).

Controls and Actions

One important property of certain controls is an event name. This is because in Windows applications, certain types of controls initiate actions, or *events*. For example, when the user clicks the OK button in a dialog box, the program processes the information entered in the dialog box and closes it.

The Window Editor makes it easy for you to associate actions with these types of controls by allowing you to specify an *event name* as a property. You have the option of using any method, form, or report that is visible to your application as an event name and can even specify source code for a customized event name method from within the Window Editor.

There are many different types of controls that you can define in the Window Editor, but before going on to describe them, a few words about data forms are in order.

Data-Aware Windows

The integration of the various tools in the Visual Objects IDE provides some powerful benefits, one of which is the ability to create data windows. Data windows are *data-aware* because they "know" about the data server(s) upon which they are intended to operate.

Note: The OLE data window type is also data-aware, as are all controls that are descendants of the TextControl class. For more information, refer to the TextControl Class help topic.

A data window knows about a data server by a link that you establish between it and one or more data servers. Once a data window and a data server are linked, you can actually link individual controls in the window (such as edit controls and check boxes) with fields in the data server.

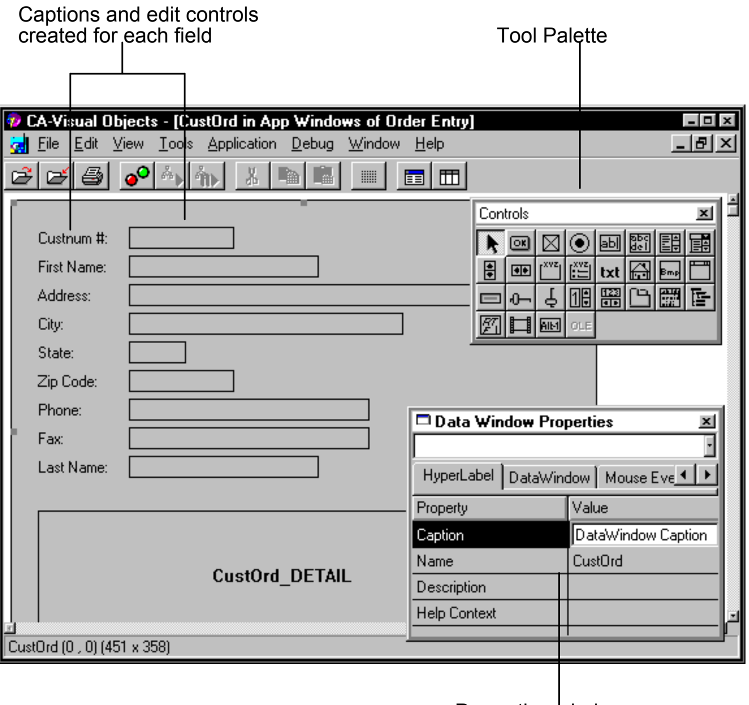
When you link a window control to a field, you are actually linking it with the field specification associated with that field—the control, therefore, automatically inherits and uses all of the field specification's properties (for example, its validation and formatting rules).

By their very nature, data windows are capable of interacting intelligently with data servers. For example, data windows can easily display the contents of a data server and have preprogrammed methods for moving among the records and manipulating the data in a data server (Go to Top and Delete Record).

Not only are data windows powerful additions to your applications, they are also easy to create. Using the Window Editor's Auto Layout feature, you can quickly link a data form with one or two data servers, creating either a single-server or master-detail data form.

When you use Auto Layout, Visual Objects automatically creates a fixed text caption and edit control for every available field in the associated data server(s). (See Types of Controls later in this section for details about these controls.)

Here is an example of a data-aware form created using the Auto Layout feature:



Data Dialog Windows

In addition to Data-Aware windows you can also create a data dialog window. A data dialog window is a window that combines features from both data windows and dialog windows. This combination allows the creation of modal data-aware windows.

Types of Controls

The tool palette in the Window Editor contains a host of buttons representing different controls. (If you prefer, the Window Editor also features an Edit Select from Palette menu command, which allows you to place controls by choosing commands from a menu.)

Note: For detailed information about the new IP address, month calendar, date time picker, and ComboBoxEx common controls, as well as the new data list view custom control, see the online help.

The following is an overview of some of the various types of controls you can create (they are listed in alphabetical order).

Animation controls are used to display silent Audio Visual Interleaved (AVI) clips. These controls are useful when a lengthy operation is being performed. They can be displayed until the operation is completed.

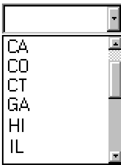
Check boxes indicate a set of options that are either on or off. If more than one check box is present on a window, the user can select as many as are applicable. The state of a check box is indicated in the box to its left: if there is a √ in the box, it is selected; otherwise, it is not.

You might use a check box on a data window to indicate a logical field. Checking the box would indicate a value of TRUE, while unchecking it would indicate a value of FALSE. For example:

☑ Shipped

Combo boxes are list boxes with a single-line edit control attached at the top. The user can either type a value directly into the edit control, or click on the Down arrow button to the right to open a list box from which to make a selection. The selection is used to fill in the edit control, which can then be edited.

In a data window, you can use a combo box instead of a list box when the field value has more possibilities than you care to list. By placing the most commonly used values in the associated list box, you give the user a quick way to make a selection, without removing the flexibility of entering values that are not listed, as shown here:



Edit controls present a blank area on a window into which the user can enter data from the keyboard. They come in two varieties, as shown below: single-line for entering one line of text, and multi-line for entering several. The user can edit the text in an edit control with the normal mouse and menu commands.

Edit controls are probably the most commonly used controls on data windows and are often used to represent fields into which the user may type almost any value.

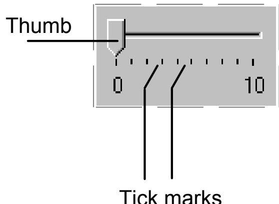
Fixed icons are graphic pictures that can be placed anywhere in a window. They are created with any graphics application, including the Visual Objects Image Editor. An example of a fixed icon is the one displayed on the Confirm dialog boxes:

Fixed text displays a caption or label anywhere within a window. A common use of this type of control is to create a caption for a single-line edit control, a feature that is utilized by the Window Editor's Auto Layout feature. For example:

First Name: Kathy

Group boxes visually indicate a set of related controls. They provide a caption to describe the controls, but serve no other purpose. They are most often used to display a group of related check boxes. On a dialog box, for example, you might give the user the option of choosing several styles for displaying text:

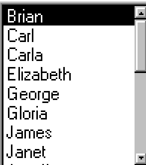
Horizontal sliders, or *trackbars*, are used to select a specific value or set of consecutive values in a range of records or options. It typically includes a slider, or *thumb*, and *tick marks* that indicate the incremental values in the range. For example, Windows provides a slider control for setting the double-click speed of your mouse.



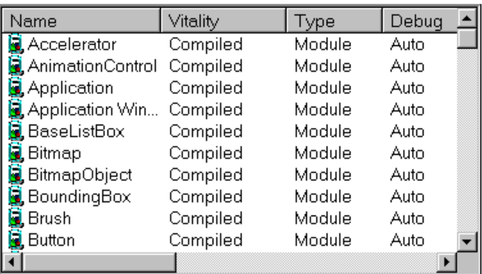
Horizontal Spinner controls consist of a pair of Left and Right arrow buttons that are used to decrement or increment a value, respectively.

HotKey edit controls enable the end user to select a valid key combination as a shortcut for performing an operation or for accessing another form.

List boxes display a list of choices to the user and allow the user to scroll through them and select one. In a dialog box, you might use a list box to allow the user to select a file name. On a data window, you might use a list box to display all possible values for a particular field. For example:



List views allow the user to view, add, delete, and arrange a list of items wherein each item consists of an icon and a label. For example, the right pane of the Repository Explorer is a list view:



The contents of a list view control can be displayed in one of four view types: Icons, Small Icons, List, and Report. Other options allow the end user to edit labels, scroll items, and select more than one item at a time.

OLE object controls (OCX) allow you to seamlessly embed other applications into the application you are currently designing. For example, if you were creating a financial application like our Order Entry sample application and you wanted to add spreadsheet capability to it, you could do so by inserting Microsoft Excel as an OLE object control.

The Window Editor is a full blown OLE container that allows the setting up of the initial state of OLE objects (how OLE objects will look when brought up at runtime). The Window Editor supports in-place and out-of-place editing of OLE objects.

Note: Some OLE server applications might not support in-place editing. In this case embedded objects will be edited out-of-place. Due to the different behavior of the servers, you may receive different behaviors on editing different types of embedded objects. It is an OLE convention that editing of linked objects is always done out-of-place. The Window Editor is also an OLE control (OCX) container.

Note: You can also *link* OLE objects to your applications. For more detailed information about linking and embedding OLE objects and OCX controls, see the Linking and Embedding OLE Objects and Controls section in the "Using the Window Editor" chapter of the *IDE User Guide* and the OLE 2.0 Features section in the "Object Linking and Embedding" chapter of the *Programmer's Guide*.

Progress bars are used to visually indicate the progress of lengthy tasks, such as installation and compilation operations. Every progress bar has two features: a range and a current position. The range denotes the length of the task from start to completion, and the current position indicates the progress made. The system uses the range and current position to calculate progress as a percentage and colors a corresponding percentage of the progress bar.

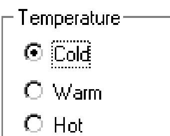
Push buttons react when the user chooses them by generating an event (see Controls and Actions earlier in this section). Some examples of push buttons are the standard OK and Cancel push buttons, shown below, used to close a dialog box or a Commit push button on a data window that commits the edits you have made.

Radio buttons behave like check boxes unless contained in a radio button group box (described below), but their appearance is different. A selected radio button contains a black dot, as shown in the next illustration.

Radio button group boxes visually indicate a group of radio buttons. Like a regular group box, they provide a descriptive caption for the controls they contain, but they have another special purpose—only one of the radio buttons within a radio button group box can be selected at any time. When the user chooses a new radio button in the group box, the previously selected one is turned off , or *deselected*.

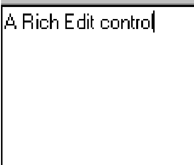
Each radio button group box behaves independently. In other words, you can place several groups of radio buttons on the same window, and the user can select exactly one radio button in each group box.

Radio button group boxes let you use radio buttons to present a set of choices to the user. For example, you might use a radio button group box on a data window to fill in a field that can only take on a limited number of values, such as a Temperature field that must be either “Cold,” “Warm,” or “Hot”:

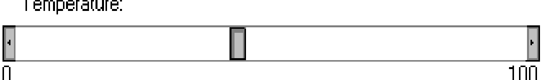


Rich edit controls allow the end user to enter, edit, delete, format, and print straight text. You can set tabs, use indentation, align and number text; and for characters, you can specify font, size, text and background color, italics, and so on.

In addition to character and paragraph formatting, you can also embed OLE objects in rich edit controls. For example:



Scroll bars display a gauge that the user can adjust using a scroll box or scroll arrows. They come in two varieties: horizontal and vertical. You could use a scroll bar on a data window to graphically represent a numeric field. For example:

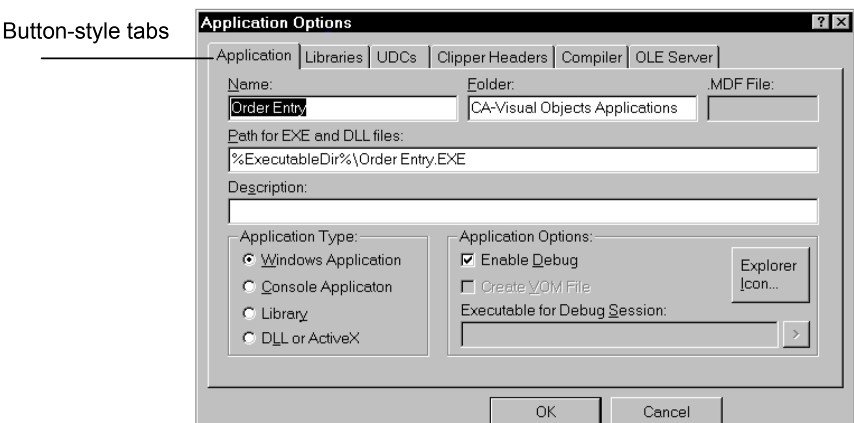


Note: The scroll bars discussed here, although visually and functionally identical, do not apply to the windows themselves, but rather to the data that the window displays. Window scroll bars (and scroll bars in list boxes and combo boxes) are handled dynamically in Visual Objects applications, depending on their current size and the amount of data that needs to be displayed.

Sub-data windows are simply data windows that you place on other data windows as controls (they are also referred to as subforms). Typically you would use a sub-data window to show a master-detail relationship between two related data servers. (We will create such a data window later as part of the tutorial in the next chapter, “Learning the Basics.”)

Tab controls are used to present data or a series of choices in a multiple-page format. It consists of one or more tabbed pages that resemble file folders. When the end user clicks on one of the tabs, the corresponding page moves to the forefront and allows access to its data and controls.

For example, the various tab pages in the Applications Options dialog box are tab controls:



Note: This dialog box has been updated in this version of Visual Objects. For more detailed information, see the New Application Options section in the introductory chapter and the online help.

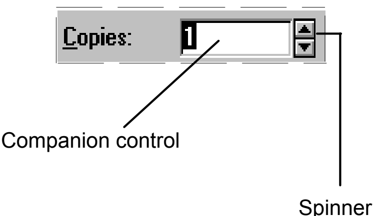
Tree view controls present the end user with a hierarchical list of items in a tree structure that can be expanded or collapsed. Each item in the tree consists of a label with an optional icon and may have an associated list of subitems. For example, the left pane of the Repository Explorer is a tree view that lists projects, applications, modules, and entities in a top-down hierarchy:

The contents of a tree view control can be displayed with buttons that expand and collapse subitems (also called *child items*) and lines that link subitems to their parent items and/or to the hierarchy's root level. Other options allow the end user to select more than one item at a time and to edit item labels.

List view controls present the end user with a non-hierarchical list of items. For example, the right hand pane of the Repository Explorer is a list view that represents the items contained within the item selected in the right hand Tree view control.

Vertical sliders, or *trackbars*, are used to select a specific value or set of consecutive values in a range of records or options. It typically includes a slider, or *thumb*, and *tick marks* that indicate the incremental values in the range.

Vertical spinners—sometimes called *spin controls* or *Up-Down controls*—consist of a pair of Up and Down arrow buttons that are used to increment or decrement a value, respectively. For example, the Copies field on a standard Print dialog box is usually a spinner control:



Note: For detailed information about the properties and styles of these controls, see the Specifying Control Properties and Style Settings section of the "Using the Window Editor" chapter in the *IDE User Guide*. Also, refer to the Properties Window help topic for the relevant control.

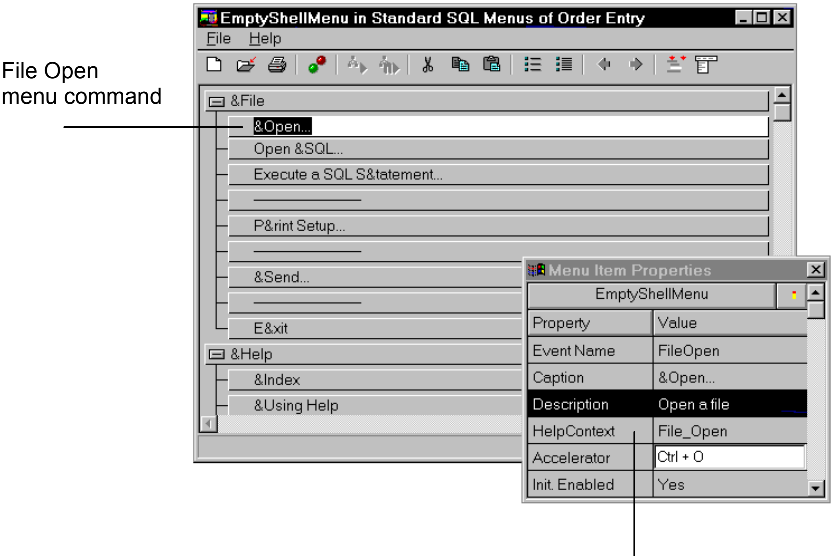
Menu Editor

The Menu Editor, shown below, provides a powerful yet easy way to create menus and toolbars for your applications.

Auto Layout

First of all, like the Window Editor, the Menu Editor features an Auto Layout feature. In the Menu Editor, however, Auto Layout is used to add one or more predefined, standard menus to an application. For example, at the touch of a button, you can add File, Edit, View, Window, and Help menus to your application. In addition, each of these predefined menus (like File) contains a set of default menu items (for example, New, Open, and Save), for which default properties are already supplied, including event names and toolbar buttons.

For example, the following shows the properties initially set for the predefined File Open menu command:



Properties initially defined for the File Open menu command

Note: See the Menu Properties Window and Menu Item Properties Window topics in the online help for new and updated menu and menu item properties, respectively.

Auto Layout provides a quick way to get started with your menu structures—you can use the resulting menus as is, or you can customize them as desired to fit your application. Of course, you can easily create your own custom menu structures in the Menu Editor.

Creating Toolbars

For each menu structure you create, you can enable or disable a corresponding toolbar. If enabled, you can choose which items in the menu structure should have corresponding buttons displayed on the toolbar, as well as, which graphic should be used to represent each item. In addition, if desired, you can choose the File Preview Toolbar menu command while in the Menu Editor to preview a menu structure's toolbar.

Menu Items and Events

Like some window controls, an important property of items on a menu is an event name. This is because menu items, like certain window controls, initiate actions or *events*. The Menu Editor makes it easy for you to associate actions with menu items using the *event name* property, exactly as previously described for the Window Editor.

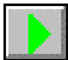
Report Editor

The Visual Objects Report Editor is a state-of-the-art report-publishing tool that allows you to create sophisticated reports. The Visual Objects Report Editor consists of the CA-Report Writer and the CA-Report Viewer.

CA-Report Writer has its own environment that you can use to create, edit, and print reports. To create a report, simply choose the Tools Report Editor menu command to start the Report Editor; you can also view your report in the CA-Report Viewer.

When creating a report you will enter a report name, choose a *data server* (DB or SQL) as well as a *report style.* The CA-Report Writer provides the following report styles: tabular, form, label, letter, free style, and cross tabular.

Based on those choices, CA-Report Writer lays out an initial report, which can then be customized as desired. For example, you can Insert a picture by using the Insert Picture menu item.



You can also preview the report by clicking on the preview toolbar button. The CA-Report Viewer displays the report that was just created. For example:

CA-Report Writer's initial report can be customized by adding report details like literal, database, and computed fields; spreadsheet-like functions; and text. You can also add graphics using CA-Report Writer's drawing features. Typeface, color, and size can be selected for your report text, and you can add bolding, italics, and other print features.

Once a report has been designed and saved on disk as a .RET file, Visual Objects automatically generates object-oriented code that you can use to access the report (for example, print it or allow the user to preview it on screen) from within your application.

CA-Report Editor makes it easy to incorporate professional-quality reports into your Visual Objects applications. Since Visual Objects also includes the royalty-free CA-Report Writer Runtime, you can deliver this powerful report technology free with any of your applications.

Note: Refer to the *IDE User Guide* and the *Programmers Guide* for more in-depth information about using CA-Report Writer with Visual Objects.

Image Editor

The Image Editor is launched using the Tools Image Editor menu command. Using the Image Editor, you can create custom icons, bitmaps, ribbons, and cursors for your applications using a drag-and-drop interface that allows you to work with several images at the same time. Icons, bitmaps, and cursors are saved in standard Windows .ICO, .BMP, and .CUR files, respectively, that can be defined to your application as resources. Ribbons are bitmap strips consisting of one or more rectangular bitmaps that have been placed side by side. Ribbons can be defined in the Menu Editor by choosing the .BMPs for the toolbar buttons.

The Debugger

The Visual Objects Debugger provides advanced tools for tracking and correcting errors that occur at runtime. With the Debugger you can:

- Control the execution of your application while viewing the source code in the 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.
- Stop program execution using breakpoints.
- Interact with the editor. Preset breakpoints in the Source Code Editor. 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.
- Use the just-in-time debugging feature, which allows the debugger to be invoked when a runtime error occurs and highlights the offensive line of code, if the entity is compiled with the debug option turned on. If the debugger is not turned on, the debugger will still be invoked but the offending line of code will not be displayed. However, the call stack and variable information are available.

Note: For detailed information about the Debugger's new features, such as AutoStart debugging and DLL debugging, both of which were described in the introductory chapter, see the online help.

In addition, Visual Objects allows you to set debugging options at *any level*—for a single entity or module, or for an entire application—and to override the current default setting at the next lower level.

This means that if you have a successful, stable application and decide to add new features to it, you can save valuable time by testing and debugging only the new module or entity. It also means that, with new applications, you can debug the application piece-meal by setting the application-level debug flag on, and selectively turning off the debug flags for modules and entities that you are not currently interested in debugging.

You can also use the terminal window in order to display data to the screen. By including the Terminal Lite library in the application properties, the terminal window will display at runtime. The following functions and commands are supported in the terminal window:

- ?
- ??
- INKEY()
- WAIT
- SET ALTERNATE
- SET PRINTER
- SET CONSOLE
- SET COLOR

What's Next

This chapter has given you an overview of the Repository Explorer and the visual editors that make up the Visual Objects IDE. These tools provide an immediate means for you to examine and control your applications and will become even more useful as your applications increase in sophistication. Building upon the tour provided in this chapter, the next chapter teaches you how to use Visual Objects with a "hands-on" tutorial.

Chapter 5

Learning the Basics

The best way to learn about Visual Objects is to use it, and this chapter lets you do just that by presenting a series of hands-on lessons that step you through the creation of a standard MDI application for order entry purposes.

You will create the order entry application through completion of the following tasks:

- Build and execute the Standard Application to explore its built-in features

  Visual Objects makes it easy to start developing applications by providing predefined frameworks with generated object-oriented code. One GUI framework, for example, is the *Standard Application,* which is fully functional and will serve as the basis for the order entry application that you will create.

- Define data servers for the new application

  A data server provides a way to interact with a database. You'll create data servers by importing a library that contains a predefined data server and associating the library with the new application. You'll then use the DB Server Editor to add a second data server to that library.

- Create a data window using the Window Editor and link it to the data servers that you created

- Add two new methods to display the data in either numeric or alphabetical order

- Customize the menus provided by the Standard Application to add three new menu items: one to open the data window created earlier and two more to implement the methods for switching between orders
- Build and execute the customized order entry application

Working through this chapter, you will create a full-featured application—complete with menus, toolbars, status bar, and event and error handling—with a minimum of time and effort. You will also gain an understanding of the skills and concepts needed to get a quick and productive start with Visual Objects.

This tutorial assumes you have read through the first part of this guide, and describes only those features of the IDE that are required to create the sample application. For a complete description of all Visual Objects features, please refer to the *IDE User Guide* and the online help. For more information on the programming concepts introduced here, refer to the *Programmer's Guide* and the online help.

This tutorial is lengthy, but you can stop at any point you like. Just make sure you save your work and shut down any editors that are currently in use. You can pick up where you left off at another time. You may find the beginning of each new lesson a convenient point for taking a break.

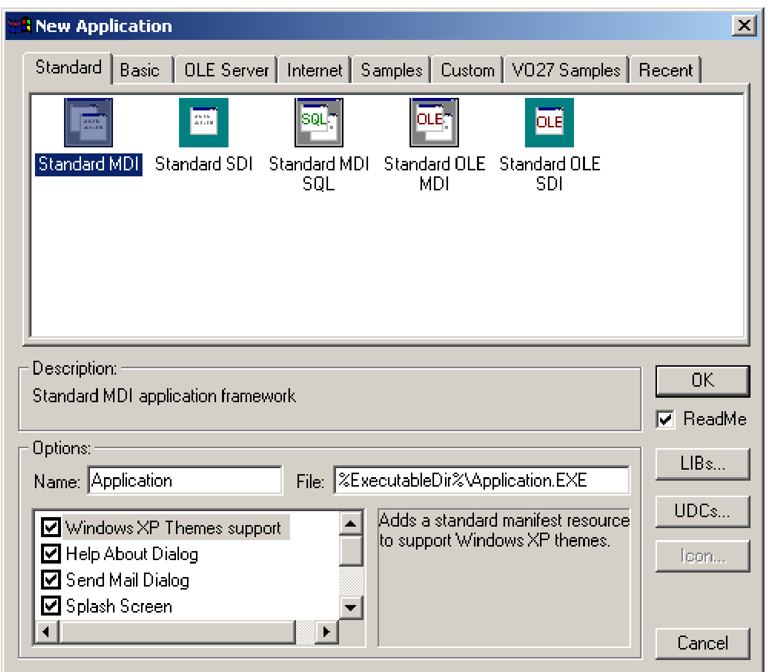
Lesson 1: A Tour of the Standard Application

There are many types of applications for windows and Visual Objects is capable of creating them all. The Application Gallery is the starting point for all new development because it presents you with an easy to use selection of pre-defined frameworks that are designed to offer an easy kick-start to your development.

Application Gallery

Open the Application Gallery by –

- Selecting the New Application… option from the File menu
- Pressing Crrl+N on the keyboard
- Clicking on the New icon on the toolbar
- Right-click on the Default Project icon in the left hand tree and click on the New Application option that is displayed in the popup-menu that is displayed.



Across the top you can see eight tabs, each of which will present you with different options but each is self-contained and presents you with all of the available options to complete the setup. These tabs provide an easy way of grouping similar items together making them easy to find.

The names of the tabs make it obvious as to the type of applications that can be generated from them. The *Samples* and *VO27 Samples* tabs are of particular interest and it is recommended that you take the time to install and examine the contained applications.

NOTE: The applications on the *VO27 Samples* tab are mostly to demonstrate new features that are only available on Windows XP using Visual themes.

As you click on each of the icons a short description of the application or sample is displayed in the *Description* area so as to give you an idea of what it will do.

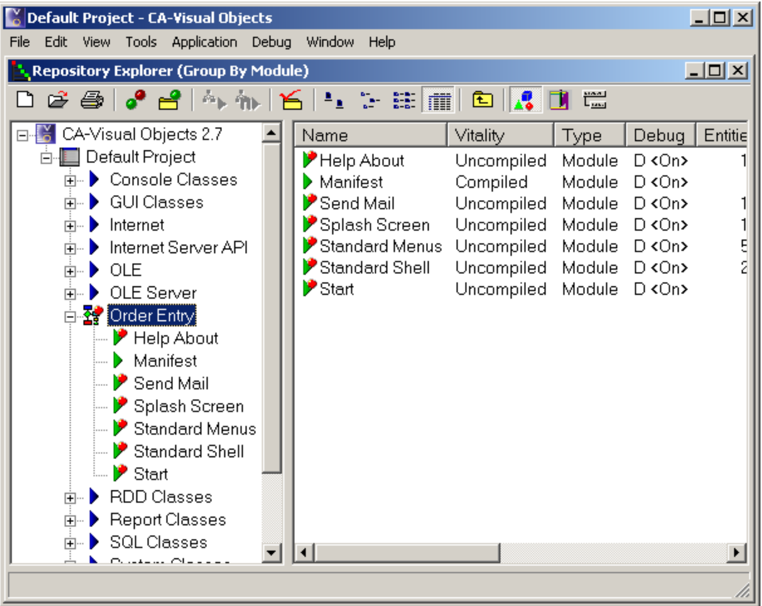
Beneath the *Description* area are the *Options* that can be set. The *Name:* is the name that will be displayed in the left hand tree of applications. The *File:* is the name to use when generating the executable file and the location to create it in. Notice the use of *%ExecutableDir%* which is a convenient way of specifying the C:\Cavo27\Bin directory or where ever you installed Visual Objects.

The last part of the *Options* section is the optional components list where you can select the required parts that can apply to the selected application. Again, as you select each component in the list, a short description is displayed beside it so that you can make a more informed decision as to whether you want it or not. In the end, those items that are ticked will be included.

Standard Application

Make sure that the *Standard MDI* icon is selected, change the *Name:* to **Order Entry**, change the *File:* to **%ExecutableDir%\OrderEntry.EXE** and press the **OK** button.

When the New Application window closes double-click on the icon beside **Order Entry** and you will see the modules that have been imported.



Building and Running the Standard Application

The few steps that you have just taken to generate this application have given you a practical and useful starting point. Without writing even one line of code, you have a working, MDI application. To fully understand this application, we will take a much closer look at the generated source code, but to start with lets see it running.

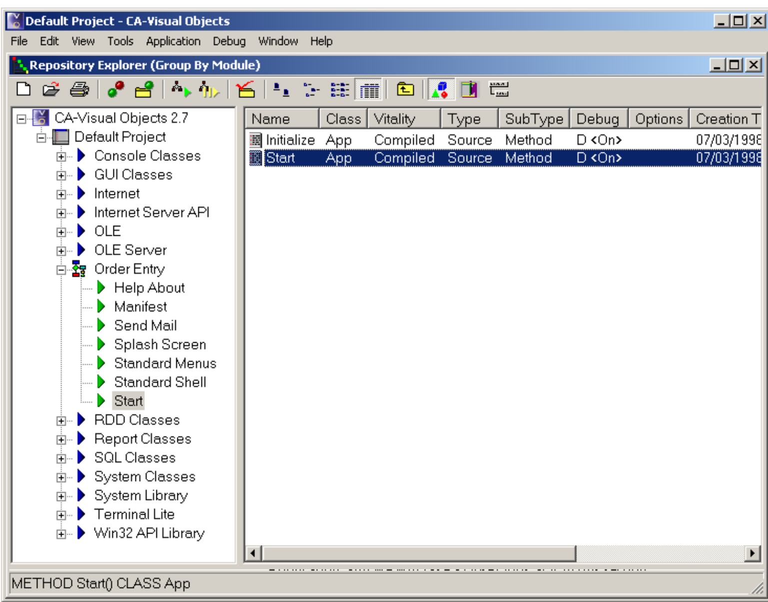
Click the Build toolbar button to compile the entire application. The dialog box that is displayed during the compilation allows you to monitor the progress. When the compile is completed without error the Build Done message is displayed in the Status Bar at the bottom of the IDE.

You have created your first Visual Objects application with just a few mouse clicks and in just a few minutes! After the build is complete, click the Execute button to run the application:

After a few seconds, Visual Objects opens a new application window titled "Standard MDI Application." Later in this lesson, we will run the application again and explore it more fully, but for now, choose the File Exit menu command, and we'll take a look at customizing the application a bit.

A Closer Look at the Application

All applications need a start method so this is probably the best place to begin. Click on the Start module in the tree and you will see two entities in the list, both of which belong to the App class. The Start() method of the App class is usually referred to as the App:Start() method.



When we were selecting the application from the gallery we selected the Standard MDI and not the Standard SDI application. Now is probably a good time to discuss the differences between these two types of application. This will also help you to understand some of the basic concepts underlying the structure of an MDI application.

MDI Application Structure

The Windows

MDI applications are structured around the presentation of multiple windows. They typically use a *shell window* as the main, or "owner," window. The documents that are opened in the shell window are referred to as *child windows*. Child windows are owned by the shell.

In the Standard Application, the shell window is created using the StandardShellWindow class. This class inherits from the ShellWindow class in order to add more functionality to the basic shell window, while preserving what is already there (for more information on inheritance and subclassing, refer to "Object Oriented Programming Concepts" in this guide). Thus, characteristic of most MDI applications, the Standard Application defines a shell window in which you can open any number of child windows.

When we speak of child windows, we are referring to windows that are owned by the shell window. These windows are typically derived from one of two classes: ChildAppWindow or DataWindow.

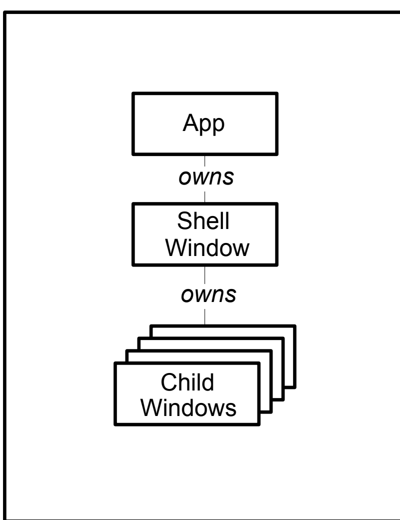
The DataWindow class actually inherits from the ChildAppWindow class and adds functionality to support linking the window directly to a database through a data server. Thus, a data window is a kind of child window. We, therefore, use the terms "child window" and "data window" interchangeably in this discussion.

Note: An SDI application, by contrast, is structured around displaying a single document at a time. Typically, it uses a top window (based on the TopAppWindow class) as the owner window, and a ChildAppWindow or DataWindow as the child. The difference is that with a top window as the owner, only a single child window can be open at a time.

The App Object

Just as each child window has the shell window as its owner, everything in a Visual Objects GUI application has an owner. This is a very important concept that controls much of the action in the application.

At the highest level, the topmost window in an MDI application—the shell—is owned by the *App*, an invisible object that controls the basic event processing of the system:



The App object represents the overall application—it starts, runs, stops, and handles all events (such as mouse clicks and keystrokes) in the application.

The App:Start() Method

Every application you create in Visual Objects requires some function or method named Start() that serves to get the application started; in the Standard Application, this is the App:Start() method.

Double-click on the icon beside Start in the right hand list to open the Source Code Editor. You should now be viewing the source code and it should look like this:

```
METHOD Start() CLASS App
    LOCAL oMainWindow AS StandardShellWindow

    SELF:Initialize()

    oMainWindow := StandardShellWindow{SELF}
    oMainWindow:Show(SHOWCENTERED)

    SELF:Exec()
```

What this does is declare, create, and show the shell window (it's called *oMainWindow* and is instantiated using the StandardShellWindow class described in the next section). It also calls the App:Initialize() and App:Exec() methods—actually, the source code reads "SELF:Initialize()" and "SELF:Exec()" but that is easily explained.

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

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

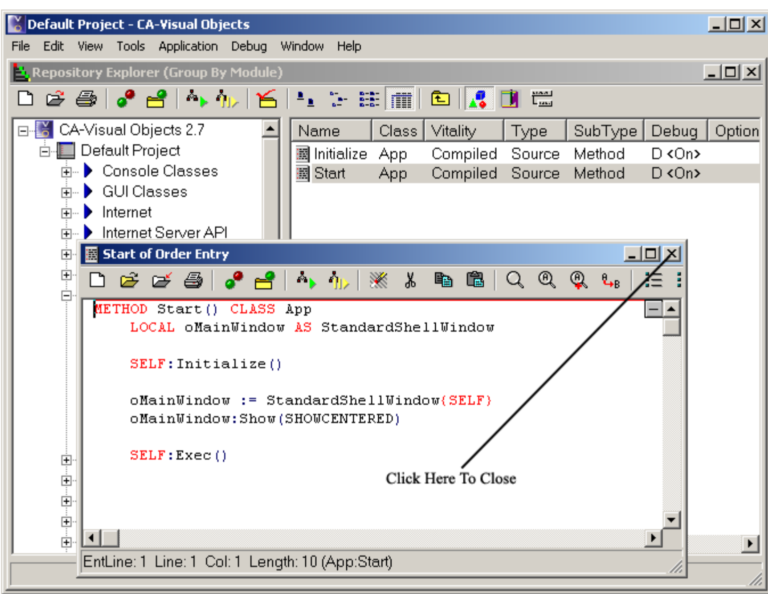
Note: Within the App:Start() method you can call App:Exec(). Doing so invokes the Windows event handling loop for your application, as well as the Visual Objects event handling mechanism. Unless you invoke App:Exec(), the system cannot start sending you events. (See Default Event and Error Handling below.)

The Shell Window

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

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

1. Click on the Source Code Editor's Close button to close this window:



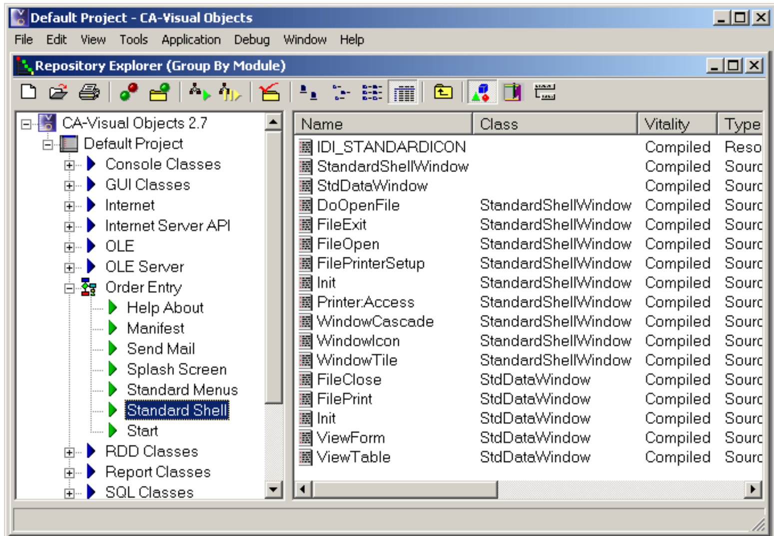
You are returned to the Repository Explorer.

2. If you have changed away from the default view, click on the Group By Module toolbar button to display the applications module listing.

3. Click on the Standard Shell module in the tree view pane of the Repository Explorer.

4. The list is shown in alphabetical order. Click in the Class column header and the list will be in alphabetical order by class.

The explorer should now look like this:



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

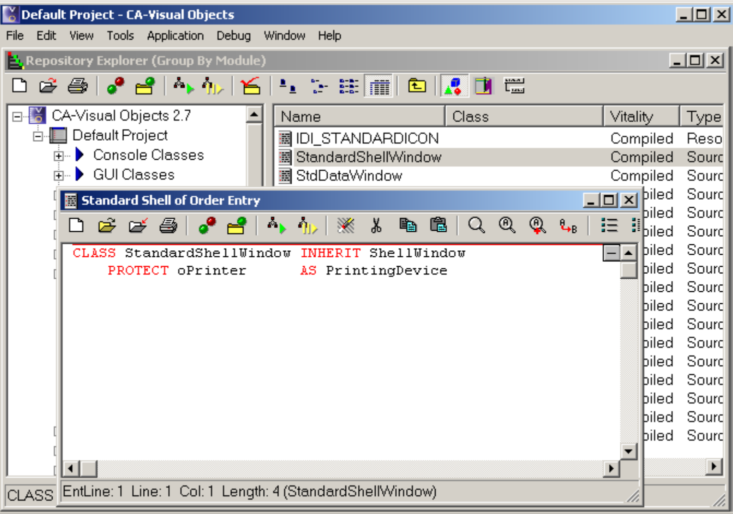
StandardShellWindow

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

```
oMainWindow := StandardShellWindow{SELF}
```

The class *declaration* for StandardShellWindow is at the top of the list view. Notice that the Class column is empty for this entity, after all, it is a class.

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



Note: The second line of code defines a PROTECT instance variable that can be used to specify printing device properties when printing from the shell window.

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

Init() Method

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

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

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

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

Standard Shell of Order Entry

```
METHOD Init( oOwnerApp ) CLASS StandardShellWindow
    LOCAL oSB AS StatusBar

    SUPER:Init( oOwnerApp )

    SetDeleted(TRUE)

#IFDEF __APPWIZ__MODDRAGDROP
  SELF:EnableDragDropClient()
#ENDIF

    oSB := SELF:EnableStatusBar()
    oSB:DisplayTime()

    SELF:Menu := EmptyShellMenu(SELF)

    SELF:Icon   := Icon(ResourceID(IDI_STANDARDICON, _GetInst()))
    SELF:IconSm := Icon(ResourceID(IDI_STANDARDICON, _GetInst()))

    SELF:Caption := "Standard MDI Application"

    oPrinter := PrintingDevice()

    RETURN SELF
```

EntLine: 1 Line: 1 Col: 1 Length: 26 (StandardShellWindow:Init)

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

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

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

As you scroll through the rest of the method, you can see that there is some code to enable the window as a drag-and-drop client; to enable a status bar and put some information on it; to attach a menu to the window (more on this below); to define an icon for the window when it is minimized (this icon is also defined in the Standard Shell module as the IDI_STANDARDICON resource and constant); and to initialize the printing device instance variable.

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

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

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

2. Change it to:

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

Close the Source Code Editor (just double-click on its system menu) and save your change by choosing Yes when prompted. Notice that in the entity list view the Vitality status has changed to Uncompiled and the Last Touched date has been updated to reflect that the source code for this entity has been changed.

Other Methods

When you return to the entity list view, you can see that besides the Init() method, StandardShellWindow has methods to respond to events generated by its menu, including one called FileExit() to shut down the application (by calling the SELF:EndWindow() method which internally invokes App:Quit()) when the user closes the shell window.

Note: While viewing the Start() method you can right-click on the StandardShellWindow text. A local GOTO pop-up menu displays, listing the Class and Init() method:

```
GOTO: CLASS StandardShellWindow INHERIT ShellWindow
GOTO: METHOD Init( oOwnerApp ) CLASS StandardShellWindow
```

Clicking on these entries will bring you right into the Source Code Editor for the Class definition and Init() method.

The Empty Shell Window

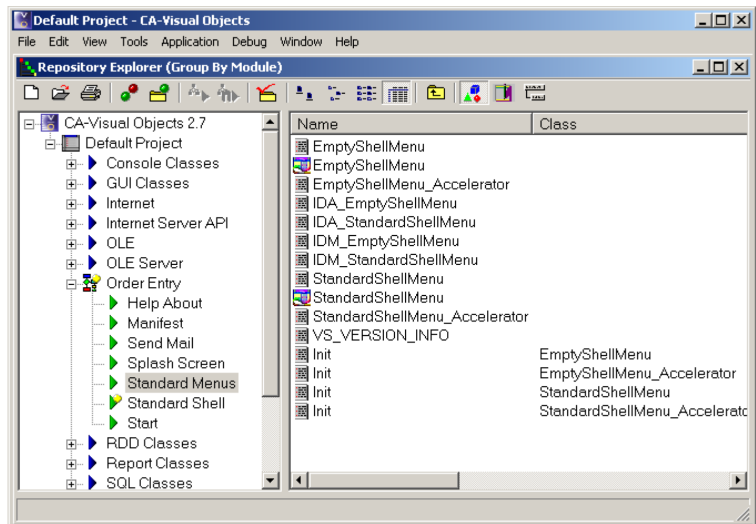
When no child windows are open in the shell window, it is referred to as the *empty* shell window. This is the state of the shell window when it is initially created by App:Start(), and the menu associated with the empty shell window is necessarily sparse because of the limited actions that you can perform when no data is present to manipulate.

EmptyShellMenu

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

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

This EmptyShellMenu class is defined in the Standard Menus module. Click on the Standard Menus module to access the entity list view for this module and click on the Class column header to sort the items:



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

Note: To display all entities the Show all items must be selected in the options Entity view tab. The options dialog box can be displayed by selecting the View Options menu command.

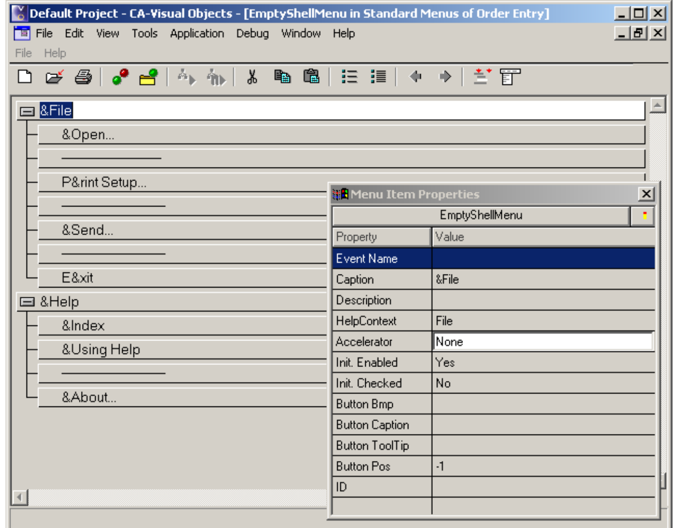
Resource Entities

For example, to define the empty shell menu and its accelerator keys in a way that is understood by Windows, there are two resource entities, IDM_EMPTYSHELLMENU and IDA_EMPTYSHELLMENU. These can be found in the list view by looking at the Type column, (you may need to scroll across the list view to see them), which specifies the entity type as a Resource and the SubType column which further specifies the resource entity as a Menu or an Accelerator. You can look at these if you like; they contain source code in the format that the Windows resource compiler wants.

The Menu Entity

The actual menu entity contains information that allows the Menu Editor to graphically display the menu layout so that you can edit it. For example, EmptyShellMenu contains File and Help menus that are used to open child windows, set up printers, exit the application, and get help.

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



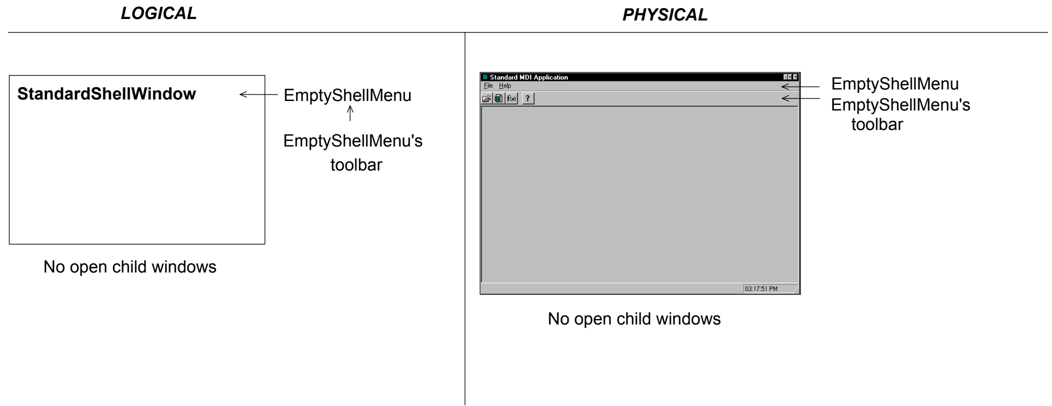
As you can see, EmptyShellMenu has just two menus: File and Help. Note that the Menu Editor allows you to design a toolbar to be associated with a menu. As part of the Standard Application, EmptyShellMenu is paired with a toolbar—when you later attach EmptyShellMenu to a window, the toolbar is automatically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

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

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

The Standard Shell Window

The relationships between the shell window, the EmptyShellMenu, and its toolbar are summarized in the diagram below:



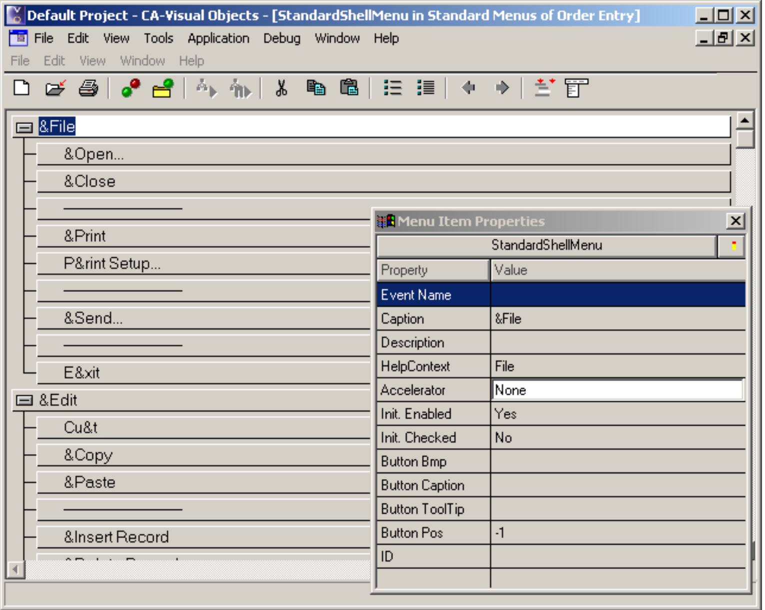
Opening a document (or child window) into the empty MDI shell window alters the nature of the shell window. It is no longer empty, but now holds another window in which data can be viewed and manipulated. When it contains one or more open documents, the shell window is referred to as the *standard* shell window. And, since the documents show data, more functionality is needed to view and manipulate the data—just the File and Help menus are no longer sufficient.

StandardShellMenu

In the Standard Menus entity listing, you will see another menu entity named StandardShellMenu. Like EmptyShellMenu, this menu also has classes, resources, constants, and methods that help define it.

If you double-click on the StandardShellMenu binary menu entity, you can see that it contains not only File and Help menus, but several additional menus and commands that let the user manipulate the database in the child window.

For example, if you use the scroll bar, you can see standard Edit menu commands like Cut, Copy, Past as well as Insert and Delete Record:



Also, like EmptyShellMenu, StandardShellMenu has its own toolbar. (When you are finished looking at the menu, close the Menu Editor by double-clicking on its system menu.)

Relationship to Child Windows

It is important to note that the shell window is not the owner of the StandardShellMenu. Instead, this menu is owned by the child window that is currently open and selected (or has "focus"). It is the child window that contains the data, and, therefore, it is the child window that requires the additional functionality provided by the StandardShellMenu. When a child window has focus, the shell window automatically "knows" to replace its own menu with the child window's menu.

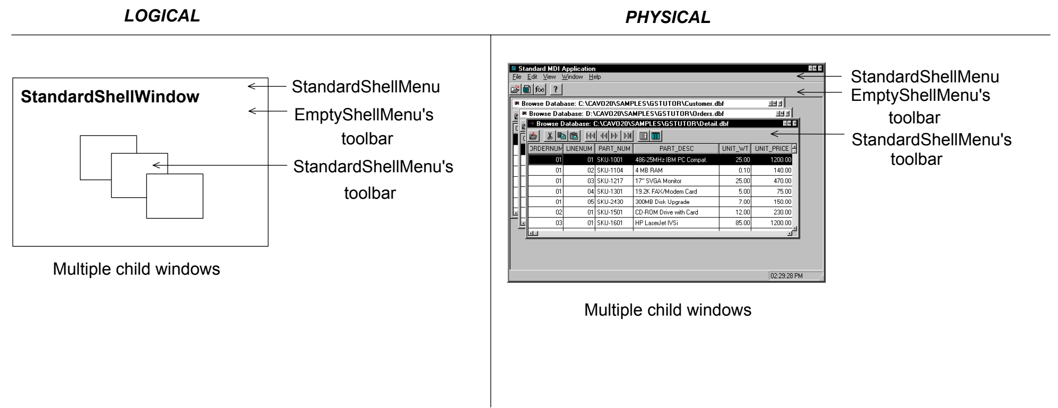
For example, you might develop a shell window into which the user can open a text editing window and a spreadsheet at the same time. In this situation, not only would the menu for each type of child window differ from that of the empty shell window, but also the various types of child windows would most likely have menus that were different from one another. The shell window knows to display the appropriate menu, depending on which child window has focus.

The Standard Application takes care of this for you—when you open a child window for a database file into the shell window, the menu displayed in the shell window is repainted, so that instead of EmptyShellMenu, the StandardShellMenu is displayed.

The replacement of one menu by another is accomplished through methods in the StandardShellWindow class. Basically, the menu selection to open a database file from EmptyShellMenu triggers an event that calls the StandardShellWindow:FileOpen() method that, in turn, calls a series of methods that instantiate the StdDataWindow class for the chosen database file.

The Init() method of the StdDataWindow class instantiates the StandardShellMenu class to display this new, more functional menu. (The StdDataWindow class is discussed in more detail in the next section.)

The relationships between the menus and toolbars displayed in the standard shell window and its child windows are summarized in the diagram below:



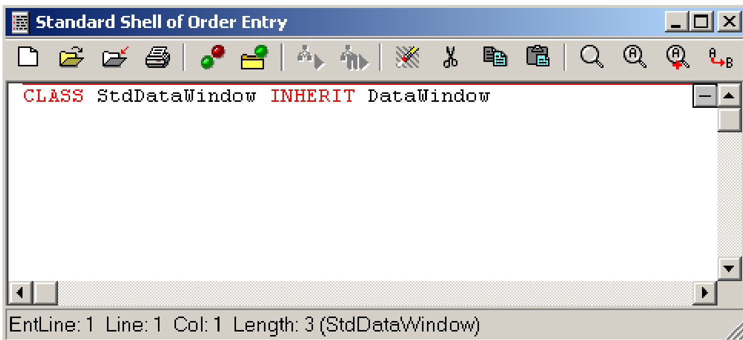
Note that while the shell window replaces its menu with that of the child window, it does not replace its toolbar. Instead, the shell window keeps its initial toolbar (EmptyShellMenu's), and each child window has its own toolbar (StandardShellMenu's).

The Child Windows

As mentioned in the previous section, the child windows opened in the shell window are instantiated (albeit indirectly) by the StandardShellWindow:FileOpen() method using a class named StdDataWindow that is defined in the Standard Shell module.

StdDataWindow

If you access the entity listing for Standard Shell and double-click on the StdDataWindow class entity, you will see from the source code displayed that it derives from the DataWindow class:



Like the ShellWindow class, DataWindow is defined in the GUI Classes library.

Data Windows

A *data window* is a special kind of window that is capable of interacting intelligently with a database. The data window can easily display the contents of the database and has preprogrammed methods for moving among the records and manipulating the data (for example, Insert and Delete Record). In fact, the StandardShellMenu gets much of its functionality by directly invoking methods defined in the DataWindow class.

Data Servers

A *data server* is an object-oriented interface provided for interacting with a database. It is through the data server that the data window connects to a database and learns about its structure.

Although you can design both data windows and data servers for specific databases using editors built into the IDE (as you will see in subsequent lessons in this chapter), the Standard Application allows you to open *any* .DBF file by instantiating it as a data server (using the DBServer class defined in the RDD Classes library). It takes advantage of the self-configuring properties of the DataWindow class to design a data window "on-the-fly" for that data server. It is the behavior that is built into the DBServer, and DataWindow classes, rather than anything remarkable done by the Standard Application, that makes programming the self-configuring data windows so easy.

.DBF Files

In fact, the generated code is quite simple. For example, this is the flow of control for opening a .DBF file:

- StandardShellWindow:FileOpen() displays a standard File Open dialog box to allow the user to select a .DBF file name and calls its DoOpenFile() method using the resulting file name.
- DoOpenFile() verifies that it has a valid file name and then instantiates StdDataWindow using owner, file name, and data server parameters and registers this new window as a child of the shell window.
- Finally, StdDataWindow:Init() instantiates a generic data window, registers its menu, instantiates a DBServer for the chosen file and links it to the data window, and displays the new data window in browse view.

Note: You can open all of these methods in the Source Code Editor at the same time and easily follow the logic described here. After you have loaded one (FileOpen(), for example), switch back to the Repository Explorer then double-click on another entity (perhaps DoOpenFile()), and it will be added to the source code currently loaded in the Source Code Editor.

Default Event and Error Handling

In a moment, we will build and run the Standard Application so that you can see the windows and menus in action, but first it will help to understand something about the basic event handling logic that is already built into the application. However, before moving on, close the Source Code Editor.

Event Handling

To briefly explain event handling, we'll start with the Windows environment. Windows provides a flexible, interactive environment in which multiple applications can be active and available simultaneously. To achieve this, all applications running under Windows interact with Windows—and possibly other GUI applications—through a *message queue*.

The message queue receives *messages* from both the operating system kernel and other applications. These messages are used to notify applications of *events* that require attention. For example, if the user presses a button or selects a menu in an application, an event is posted to the message queue. (The Windows message queue maintains both user-generated and system-generated events.)

The message queue then notifies the various applications of the events that pertain to them. When programming for Windows, therefore, your applications must know how to *handle* (interpret, respond to, and generate) events.

In Visual Objects, the basic event handling logic is this: a command event (such as a menu or push button selection) is sent first to the lowest-level window that has focus. If that window has no mechanism for dealing with the event, the event is passed up to the window's owner. This propagation of an event up the ownership chain continues until some window handles the event or until it finally reaches the App, where it most likely ends up doing nothing.

Some examples of built-in event generation and handling in the Standard Application have already been mentioned in this section. For example, in the empty shell window when the File Open menu command or the Open toolbar button is selected, a FileOpen event is generated, causing the StandardShellWindow:FileOpen() method to be invoked.

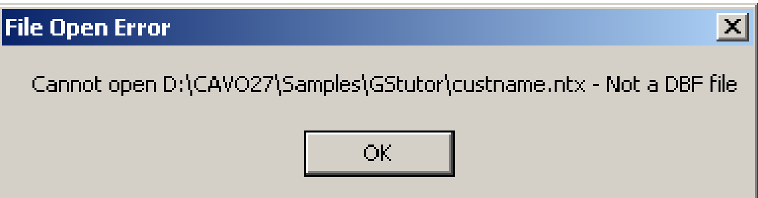
The event names for menu commands are defined in the Menu Editor, as you will see later in this chapter. Visual Objects processes these events automatically by trying to match the event name first to a method, then to a Window or ReportQueue subclass of the same name. The event is then handled either by invoking the method or instantiating an object of the subclass.

This automatic propagation is quite useful. In an MDI application, for example, File Save and File Print are normally handled by the child window because these commands are specific to each document, while File Open and File Print Setup are more general and are, therefore, normally handled by the shell window.

Again, you can see this illustrated in the Standard Application. The StandardShellMenu has both a File Open menu command and an Open toolbar button, but the child window that owns this menu does not have a FileOpen() method. So, when the FileOpen event is generated from a child window, it ends up being handled by the StandardShellWindow:FileOpen() method.

Error Handling

There is also some error handling provided by the Standard Application. For example, if you attempt to open anything other than a .DBF file with the File Open menu command, you'll receive a message similar to the following:



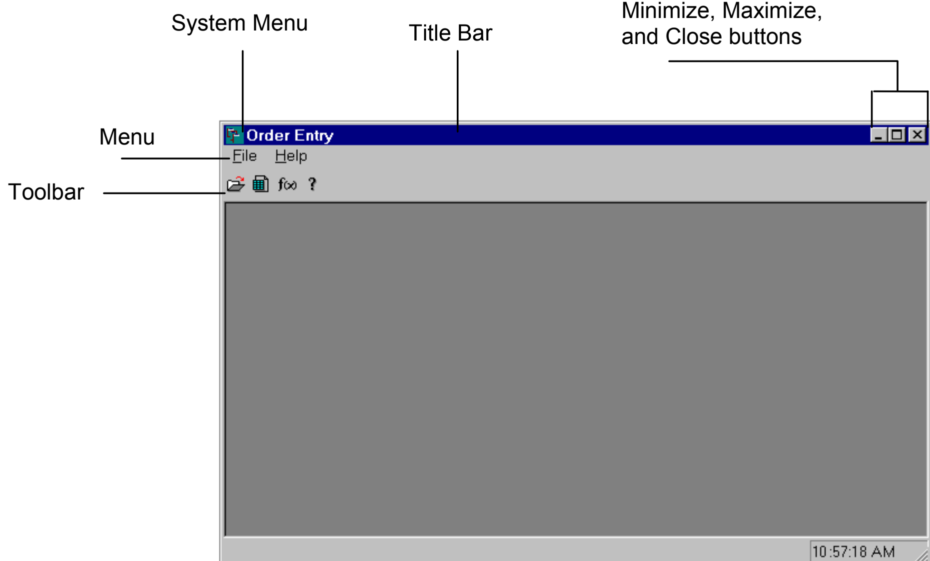
A Closer Look at the Standard Application

Now that you have examined the source code and have a general understanding of the structure of the Standard Application, we will build the application and run it again, this time taking a closer look at its features.

1. Click the Build toolbar button to compile the application.
2. After the build is complete, click the Execute button.

The Empty Shell Window

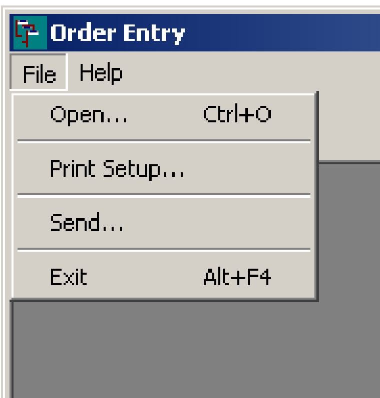
After a few seconds, Visual Objects opens a new application window titled “Order Entry”. (This bit of customization is due to the change you made earlier to StandardShellWindow:Init().) As expected, an empty shell window is displayed:



As you can see from this simple window display, the Standard Application provides a host of useful, functional application features. In addition to standard Windows features like the title bar, menu bar, system menu, minimize and maximize icons, and resizable border, this default application also includes some other handy items.

The Menus

The menu associated with the empty shell window is very sparse, as discussed previously. For example, take a look at the File menu by clicking on it or typing Alt+F:



This menu has only four commands (Open, Print Setup, Send, and Exit), which is all it needs. There is no Print command, because there is nothing to print, and there is no Close command, because there are no open child windows to close.

The Toolbar

The toolbar for the empty shell window is also brief, containing only two buttons that serve as shortcuts for the File Open and Help menu commands.

The Status Bar

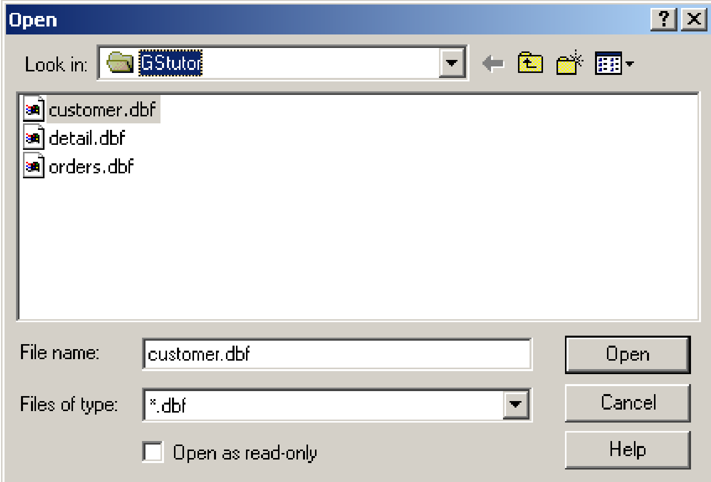
The status bar is also full of useful information. For example, highlighting the commands on a menu or moving the mouse over the buttons in a toolbar displays descriptive text about those features in the bottom left-hand corner of the status bar. The status bar also tracks the current time.

Opening Database Files

As we mentioned earlier, the Standard Application supports the ability to open any .DBF into a self-configuring data window. Let's open the CUSTOMER.DBF file and see how this changes the nature of the shell window.

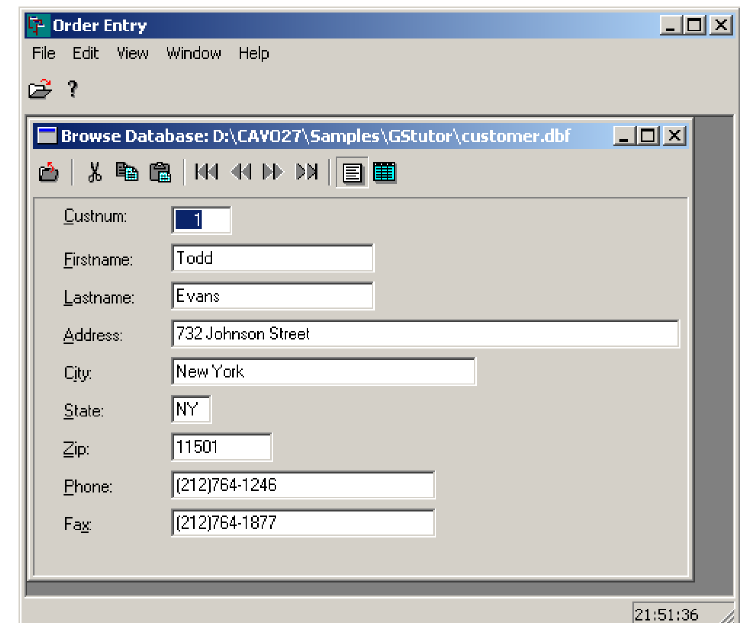
1. Click on the Open toolbar button or choose the File Open menu command.

   A standard Open dialog box is displayed:



2. Since this dialog is already preset to .DBF files, simply switch to the Visual Objects \SAMPLES\GSTUTOR directory, highlight CUSTOMER.DBF, and choose Open.

The selected database file is then opened in a data window:



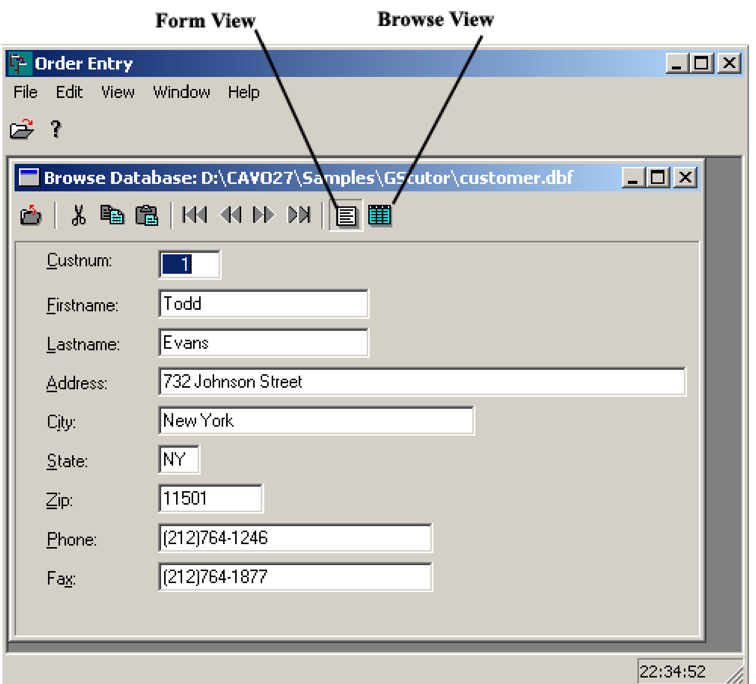
This data window allows you to view and modify the data stored in the selected database file. You can see what database file a window is associated with by the text displayed in the child window title bar.

Important! *The Standard Application does not have an option for opening index files. Therefore, if you are using it to open database files with associated index files, do not edit data or add or delete records. Otherwise, the index files will be out of date the next time you attempt to use them.*

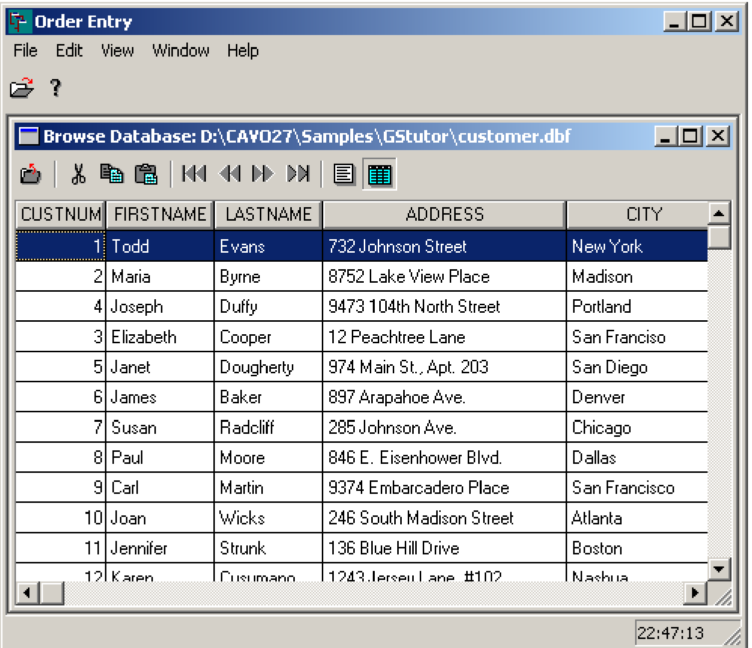
Switching Between Form and Browse View

Data windows can display the information they get from their associated data servers in two ways: in *form* view or in *browse* view. Form view displays all the fields in a record as individual edit controls (a single record at a time). Browse view, on the other hand, displays many records at once using a spreadsheet-like data browser; in this view, each field in the data server represents a column in the browser and each record a row.

Using the Form View and Browse View toolbar buttons, you can switch between these two views:



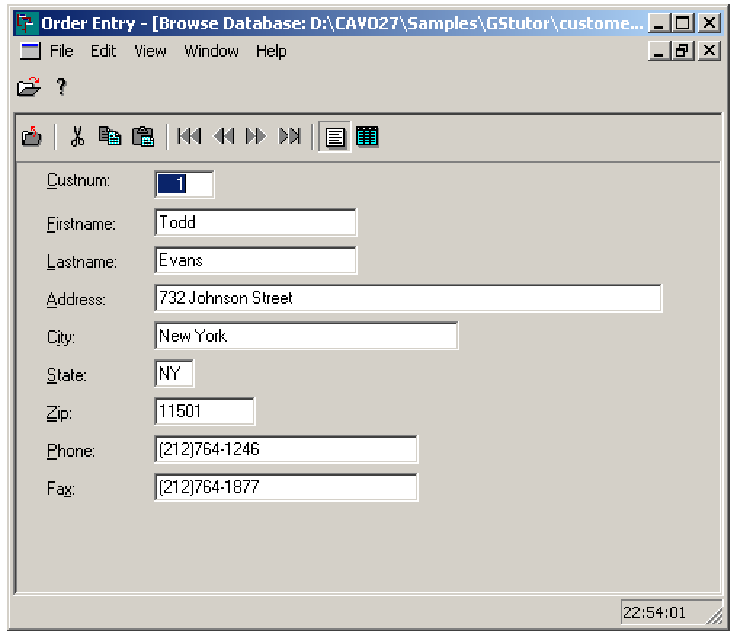
For example, click on the Browse View button in the current data window—the program redisplays the window, showing many records in browse view:



Dynamic Scroll Bars

You will notice, if you compare these two views, that the data window in browse view has both a horizontal and a vertical scroll bar because all the data will not fit into the window, while the form view may have only a vertical scroll bar.

Furthermore, if you now switch back to form view by clicking on the Form View button, and then click on the maximize button (every data window, as a child of the owner MDI shell window, has its own independent controls, such as minimize and maximize buttons and a system menu), the vertical scroll bar disappears:



The scroll bars in child windows dynamically come and go, based on the current size of the window and how much data needs to be displayed. This, along with the minimize and maximize buttons and the system menu, is a built-in feature of all child windows that requires no programming on your part. You can also resize the form view window to see the dynamic creation of the horizontal and vertical scroll bars.

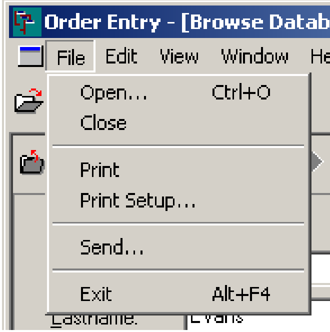
Note that in both the browse and form view, you can edit the displayed data by simply typing over the contents of the field that is currently selected. Pressing Tab "commits" the changes and moves to the next field.

The Standard Shell Window

As discussed, when you opened a child window in the shell window, the menu previously displayed in the empty shell window was automatically replaced with the menu associated with that child window.

The Menus

Let's take a look at some of the features of the new menu. First, open the File menu:



You will notice right away that this File menu has a Close command that was not present in the empty shell window's File menu. This command closes the currently selected data window.

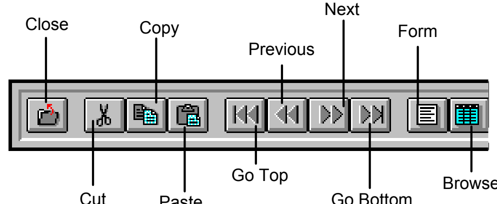
If you continue to explore by opening the Edit menu, you will see commands to navigate among the records in the current data window (such as Next and Previous), as well Delete Record and Insert Record commands, whose purposes are self-explanatory.

Finally, the View menu contains commands to switch between form and browse view, and the Window menu provides standard Tile and Cascade commands, the ability to choose between all open child windows, and a Close All command to close all open data windows.

The Toolbar

You will also notice that, while the menu in the shell window has been replaced with that of the child, each window still has its own toolbar (which you have already used to switch between browse and form view).

The child window's toolbar is illustrated below with each button identified:



The Status Bar

When a child window has focus, moving the mouse over the toolbar buttons will display the tooltips for each button and a description on the status bar. It can also be used to display a basic description for the currently selected field, but you have to provide the descriptions as part of the data server definition—it is not built into the self-configuring data window created by the Standard Application. (You will implement this feature when you create some data servers of your own in the next lesson.)

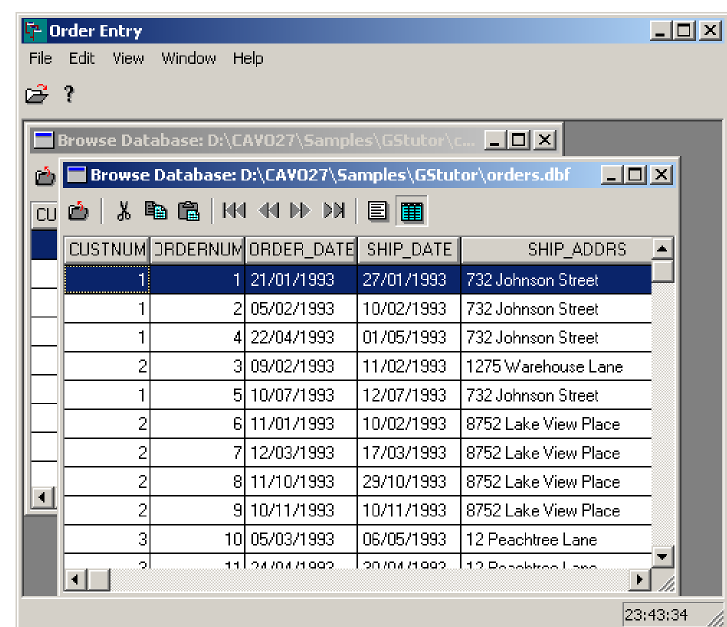
Opening Multiple Windows

Finally, to get the full flavor of the capabilities of the Standard Application, we will open a few more data windows and see what happens.

1. Return the window to browse view by clicking on the Browse toolbar button.

2. Restore the CUSTOMER.DBF window by clicking on its restore button.
3. Using either the Open toolbar button or the File Open menu command, load the ORDERS.DBF file located in the Visual Objects \SAMPLES\GSTUTOR directory.

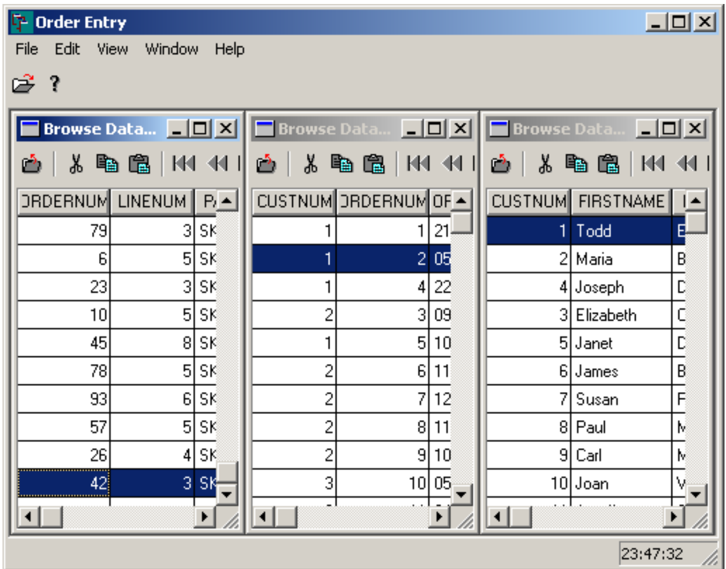
Switch to browse view, displaying the following:



4. Click the Next toolbar button to move the cursor to the second record.

5. Now, open DETAIL.DBF (also located in \SAMPLES\GSTUTOR) in another window, switch to browse view, and click the Go Bottom toolbar button to move the cursor to the last record.

6. Tile the windows using the Windows Tile command, and notice how each window maintains its own record pointer:



7. When you are through, choose the Window Close All command.

Using OLE Database Files

Visual Objects supports an OLE field type in database files. Using the OLE field type, the user can insert objects or OLE controls into the database. These fields can also be changed at runtime. Data servers that contain an OLE field can be created using the Visual Objects DBServer Editor. To display the server at runtime, a window can be created using the Auto Layout feature of the Window Editor.

At runtime, there are several things that can be done to the OLEObject field:

- Insert a new OLE object (through the edit menu, through drag-and-drop, or through Paste Special from the Clipboard)
- Insert a linked object
- Change the OLE object

The following displays a data server with an OLE field that contains a photo of the Visual Objects development team:

For more information, see the Using OLE in Database section of the “Object Linking and Embedding” chapter of the *Programmer’s Guide*.

Summary

In this lesson, you've seen and learned quite a bit about the Standard Application generated by Visual Objects. You've been given a basic description of the structure of an MDI application, and you've seen how the Standard Application implements that structure by looking at both the generated source code and the running application.

The Standard Application demonstrates many of the features available in Visual Objects. A lot of these features are fully automatic, such as:

- The minimize and maximize buttons, system menu, and title bar that are part of every shell and child window that you create
- The scroll bar behavior that is built into every child window
- The ability to open multiple documents simultaneously
- The ability to open .DBF files in a self-configuring data window
- The ability to open multiple documents simultaneously
- The event handling built into every application

Other features, such as displaying information on the title and status bars and the ability to navigate and change the contents of a database, require a minimum amount of programming because they are built into the GUI Classes library as methods and properties of the various window classes.

Still other features—such as menus, toolbars, data forms, and data servers—are facilitated by the code generators associated with the various editors in the IDE (which you will see in subsequent lessons). The IDE editors take full advantage of the automatic event handling mentioned earlier, making it easy for you to connect events to items on a menu and controls on a form, as you will also see in subsequent lessons. Finally, the editors understand the relationships between menus, toolbars, and forms of various types and, therefore, allow you to make the connections between them as part of the design process.

The Standard Application, even with all of its functionality, will seldom be enough to handle all of your business needs, but it provides a great starting point for creating a customized application. In this lesson, you've already done a very small bit of customization by changing the shell window's title bar caption, but let's move on to a more in-depth alteration.

Lesson 2: Setting Up the Data Servers

Up until now, we've called this new application the Standard Application because, after all, it has hardly anything to do with order entry yet. In fact, the only code you've written so far is to change the title bar caption of the shell window. Starting with this lesson, however, the changes you will make will be more significant; and we will, therefore, refer to the application as Order Entry from now on.

In this lesson, you will set up data servers for two database files: CUSTOMER.DBF and ORDERS.DBF. The concept of a data server was mentioned briefly in the previous lesson because the Standard Application uses them to interact with the self-configuring data windows that it creates.

Basically, a data server is a high-level, abstract entity designed to give you a consistent OOP interface for your database files and, more importantly, to allow them to interact with data windows. The data server acts as a database interpreter, defining its file (or table) name, specifying the order in which it is accessed, and providing a mechanism for extending field definitions beyond the basic name, type, and length information stored in the database file structure (this last part is accomplished using a *field specification*).

As you work your way through the development cycle, you can make on-the-fly changes to a data server (for example, changing the validation or formatting rules for one or more fields). Visual Objects ensures that these changes are reflected in all appropriate places, such as a window that is associated with that data server or another data server that is sharing the same field specification.

The purpose of this lesson is to provide the basis for creating a customized, master-detail data window in the next lesson. The information stored in these data servers will be automatically picked up by the data window, and you will see the results when you connect that window to a menu command and run the new application.

Importing the OE Data Servers Library

To make this lesson as quick and simple as possible, the data server for CUSTOMER.DBF is already defined for you. It resides in a separate library that is stored on disk as an Application Export Format (.AEF) file, rather than in the repository.

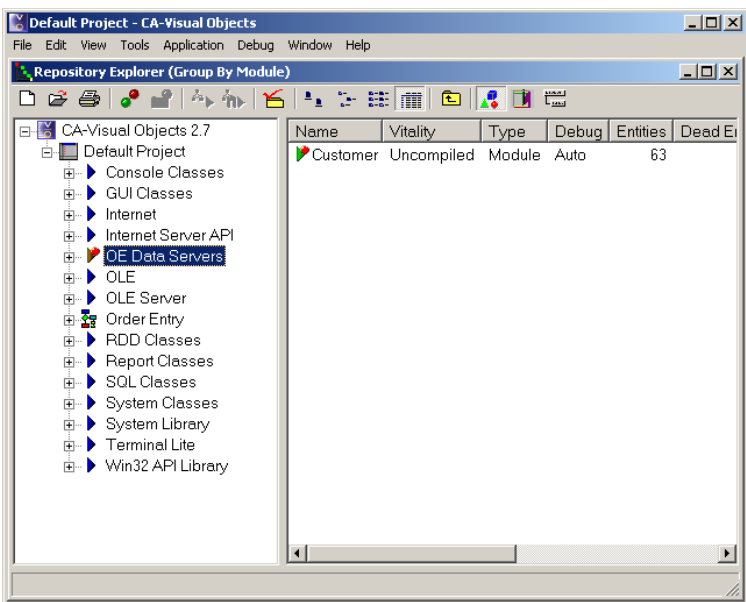
Note: The applications and libraries you create in Visual Objects can be stored external to the repository as .AEF files using the File Export command to *export* them. When desired, you can *import* them back into the repository with the File Import command.

To access the predefined data server, you need to import the OE Data Servers library as follows:

1. From the Repository Explorer click on the project that contains the Order Entry application.
2. Choose the File Import command.

   The Import Application dialog box appears.
3. Select the OESRVR.AEF file from the Visual Objects \SAMPLES\GSTUTOR directory, and then click Open.

Visual Objects creates the library and adds an entry for it (labeled "OE Data Servers") in the Repository Explorer's tree view pane:



Note: For more information about importing applications, see the "Importing and Exporting Files" chapter in the *IDE User Guide*.

A Quick Tour of the Customer Data Server

Before you move on to create a data server for the ORDERS.DBF file, it will help to take a look first at the data server you just imported. It was created for the CUSTOMER.DBF file using the DB Server Editor, the tool in the Visual Objects IDE for creating DBF-style data servers.

Loading the Customer Data Server

To load the Customer data server in the DB Server Editor:

1. Click on the entry representing the OE Data Servers library.

   Expand the tree structure to display the Customer module.

2. Click on the Customer module button to display a list of entities in that module.

   There are lots of entities in this module, but we didn't write a single line of code to create this data server. All the code you see was generated by the DB Server Editor, which you are about to launch.

   Note: To display the *prototype* of any entity (non-binary) in the status bar of the Repository Explorer, move the mouse over the list of entities in the list view. This feature of Visual Objects allows you to view information about a *source code* entity without having to open it up in the Source Code Editor.

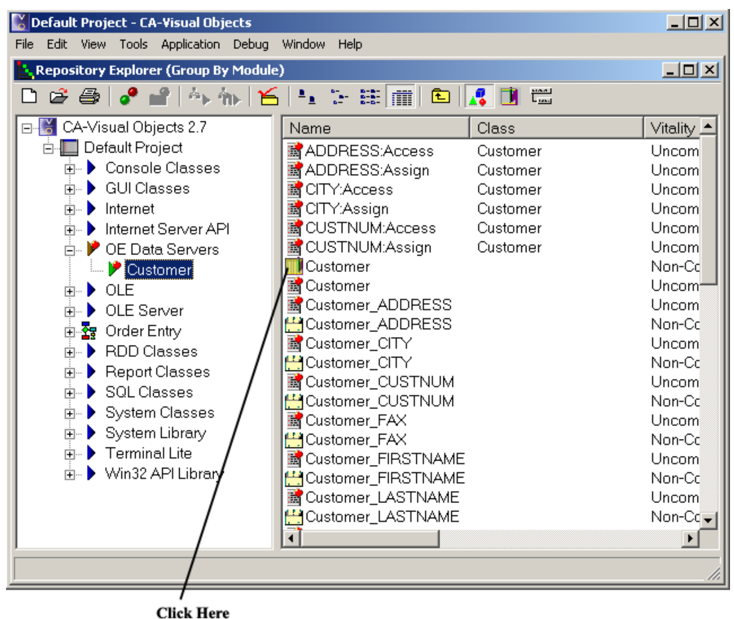
3. Look down the list view until you see the Customer *class* entity—entity types are displayed under the Subtype column. Double-click on this entity.

   This will load the source code for the Customer class declaration in the Source Code Editor. The Customer class has six instance variables. One is CDBFPath, which defines the drive and path name used to locate all files associated with this data server (i.e., CUSTOMER.DBF and all of its index files).

Note: If CDBFPath does not currently point to the correct drive or path, you can change it later in this lesson (step 6).

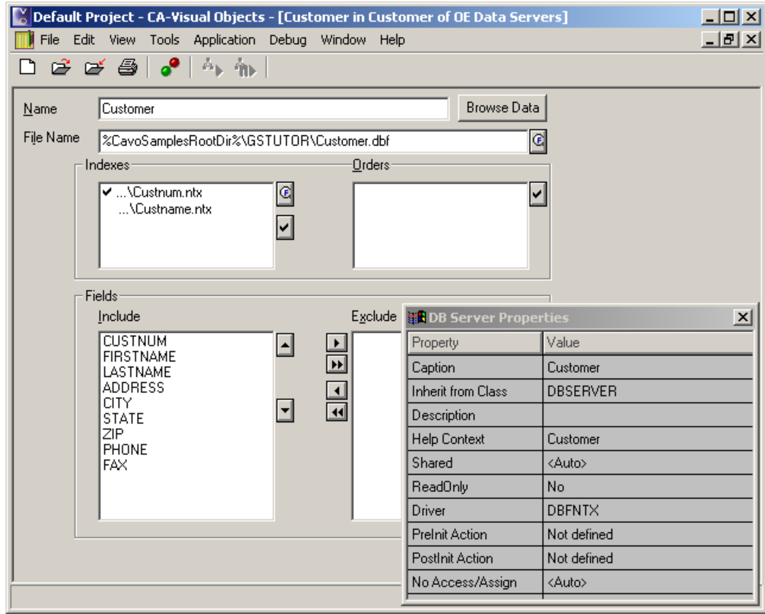
4. Close the Source Code Editor.

5. Next, double-click on the Customer server entity to load it in the DB Server Editor:



6. The DB Server Editor window appears.

   Your desktop should now look as follows after maximizing the DB Server Editor:



The first thing you will notice about the DB Server Editor is how simple its desktop is—just a few edit controls and list boxes, and the floating Properties window. (You're going to see a lot of these Properties windows—they are featured in almost every visual editor.)

The simplicity of this interface makes it easy to design and create data servers (as you will see later). For example, in the File Name edit control, you can see that this data server is associated with CUSTOMER.DBF. It also knows the location of the file, %CavoSamplesRootDir%\GSTUTOR\. This shows another of those special system variables that can be used to make Visual Objects more configurable.

Note: If you are going to put your data files in the same directory as the final executable file, you can remove the path completely because Visual Objects will automatically look in the current directory for the data.

In addition, if you look at the Properties window, you can see the various properties that can be defined for a data server, including file open modes (Shared and ReadOnly) and the name of the associated RDD (replaceable database driver).

The Indexes List Box

In the Standard Application, you could only open database files and you were warned against editing data in indexed database files because the corresponding index files were not opened. By defining your own customized data servers, you can associate as many index files as you like. You can also select the controlling order.

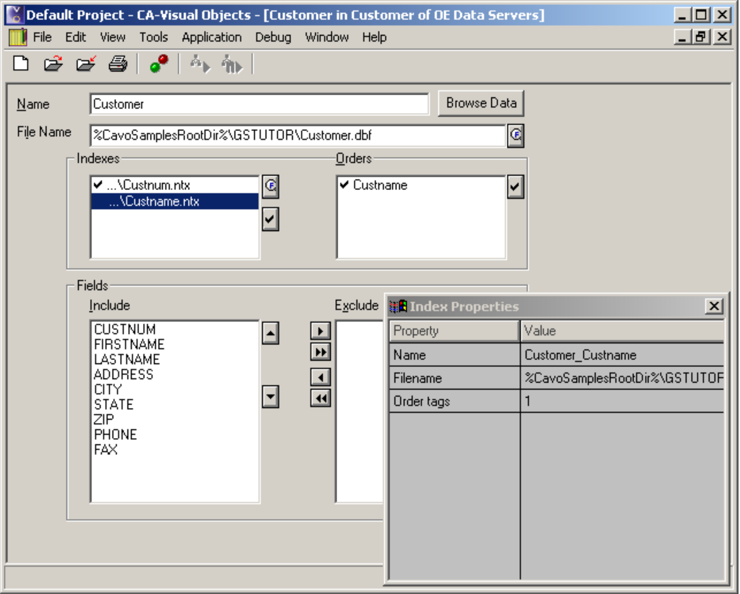
For the Customer data server, there are two index files in use, CUSTNUM.NTX and CUSTNAME.NTX. These files are listed in the Indexes list box—any time you use this data server, both index files will be updated with any changes made to the database file.

As you can see, CUSTNUM.NTX has a check mark next to it. The check mark indicates that this index file contains the controlling order.

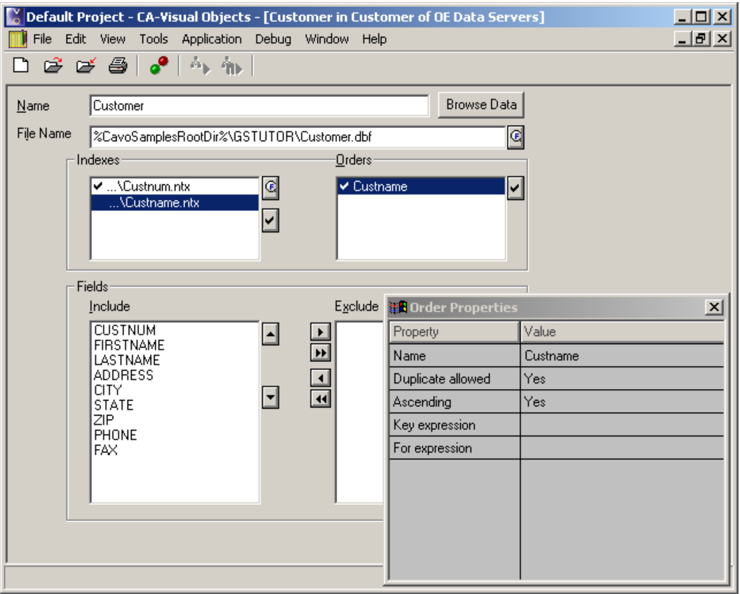
The Orders List Box

Now click on CUSTNUM.NTX.

The Properties window changes so that it displays properties related to the index file, and CUSTNUM appears in the Orders list box with a check mark. The Orders list box shows all orders defined for CUSTNUM.NTX and further defines which of them is the controlling order:



Click on CUSTNUM in the Orders list box.



Again, the Properties window changes, this time displaying properties related to the individual order.

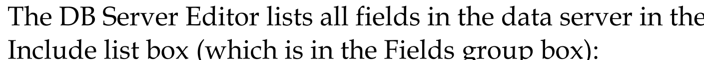
Because we are using .NTX index files, which support only one order per file, you can quickly choose a new controlling order by highlighting its associated index file in the Indexes list box and clicking on the button with the check mark icon.

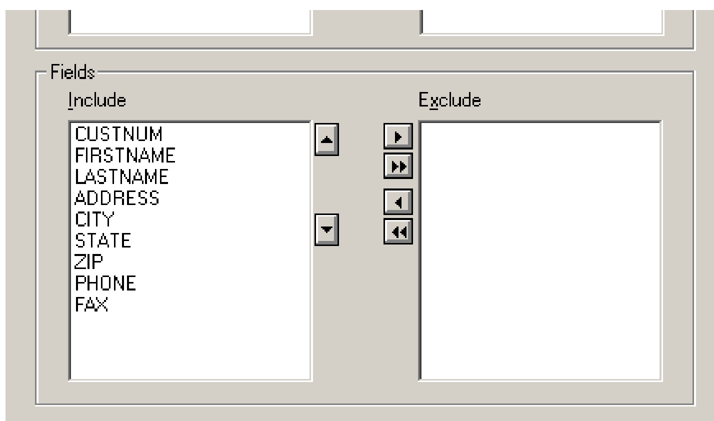
Note: If you were working with an RDD that supported multiple orders per index file, the Orders list box would show all the orders defined in the highlighted index file. Thus, to define the controlling order, you would check the index file that contained the controlling order in the Indexes list box, then check the appropriate controlling order in the Orders list box.

However, you'll add menu commands and methods to the Order Entry application later in this tutorial, which will allow the user to switch from one sort order to the other in the resulting application.

The Fields Group Box

The DB Server Editor lists all fields in the data server in the Include list box (which is in the Fields group box):





Include Versus Exclude

The Include list box is used by Visual Objects to determine which fields in the database file should be accessible when this data server is used.

If desired, you can use the Left and Right arrow buttons situated between the Include and Exclude list boxes to move one or more fields to (or from) the Exclude list box. Fields that are moved to the Exclude list box will be inaccessible when using this data server, and a reference to an excluded field in your application will result in a runtime error.

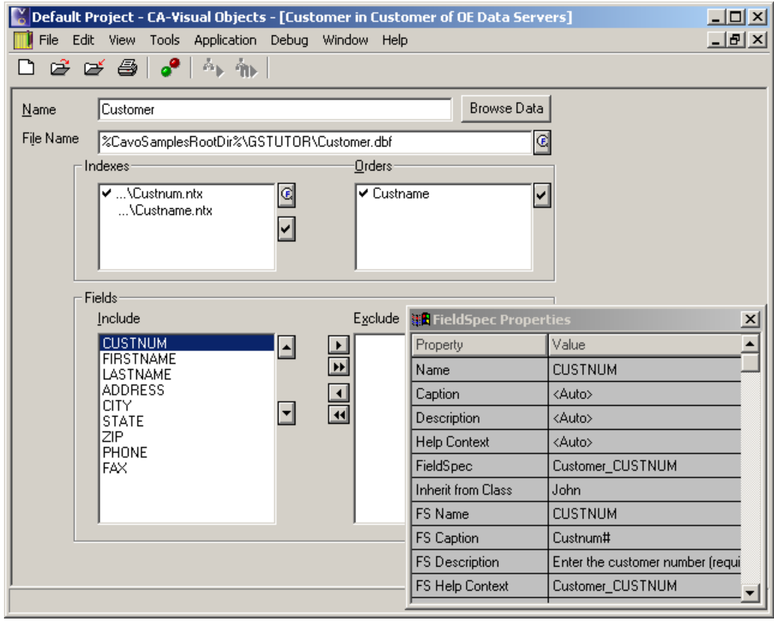
Field Order

If desired, you can use the Up and Down arrows to the right of the Include list box to reorder the fields (for example, move one in the middle to the top or bottom of the list).

The order in which fields appear in the Include list box determines the order in which they will appear when you use the Auto Layout feature of the Window Editor to define a data window for this data server (more on this later). (It also determines the order in which the fields are created if you generate a .DBF file using the File Export command.)

Field Properties

Finally, take a look at the properties available to a field. Click on the CustNum field, and watch as the Properties window changes to reflect field-related properties (and those defined for the CustNum field in particular):



Fields in a data server have many properties (such as Picture and Validation), all of which are shown in the Properties window when a field in the Include list box has focus. Note that these properties are not part of the physical structure of the underlying database file, although some of them (such as type and length) have counterparts in the file structure.

The data server uses something called a field specification (derived from the FieldSpec class) to group *all* the properties of a field into a single, logical entity, so that the information pertinent to a database field can be conveniently located. This arrangement has several advantages, but the most important ones are listed below:

- You can reuse a field specification's information in other data servers. For example, in many cases, the different data servers your application uses contain similar, if not identical, fields (for example, all zip code fields are typically the same, regardless of where they are used).
- Using a field specification, you can create just one field specification and reuse it in each data server that needs it. Thus, multiple data servers can access the same property values for common fields.
- A field specification's properties are automatically inherited by data windows. Many of the properties that you define for a field specification in a data server are designed to be used by data windows that you create using the Window Editor. Thus, you need only define the attributes once, and they will be automatically inherited and used by any data window that is linked to that data server.
- Even though it may be used in many different places (for example, multiple data servers and data windows), you can make changes to a field specification quickly and easily, and in just one place.

Changing a field's properties is easy, even if numerous data windows and data servers use the field specification. If you change a field specification (whether in the DB Server Editor, the FieldSpec Editor, or the Window Editor), the Visual Objects repository ensures that the change is automatically propagated to all other entities that use it. (You'll see this later on in this lesson.)

With that explanation, the tour of the Customer data server is concluded. Let's move on to create another data server, this time for the ORDERS.DBF file. By doing so, you'll get a better understanding of field specifications by reusing some of the ones that were already defined for the Customer data server. You will also see how changes are automatically propagated when field specifications are modified.

Before moving on, close the DB Server Editor by double-clicking on its system menu and, if you have made any changes please do not save them at this time.

Creating the Orders Data Server

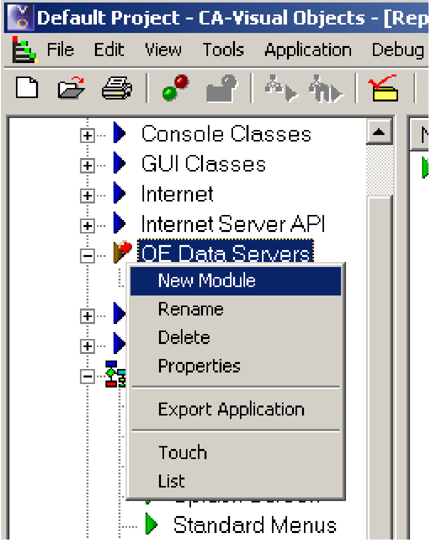
The process that you are about to go through to create the Orders data server is remarkably similar to the process we went through while looking at the Customer data server that you imported in the previous step.

In this section, you will explore some of the features of the DB Server Editor hinted at in the previous section and will get a better understanding of how this editor works.

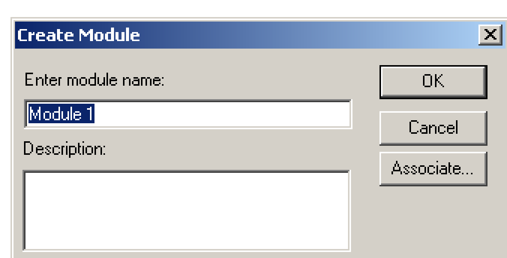
Starting the DB Server Editor

At the moment, you should have the Customer module of the OE Data Servers library selected in the Repository Explorer, perform the following steps to create the Orders data server:

1. Right click on the OE Data Servers icon in the tree view to show the pop-up menu and select New Module:



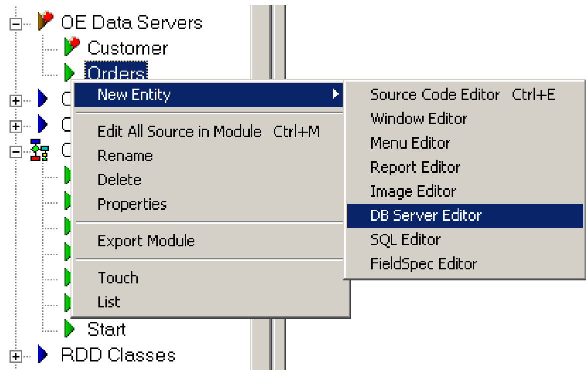
The Create Module dialog box appears:



2. In the Enter Module Name edit control, type **Orders** and choose OK.

   A new module is added to the Repository Explorer's tree view pane.

3. Select the Orders module in the tree view, and right click on its icon to show the pop-up menu:



4. Choose the DB Server Editor command from the second pop-up menu.

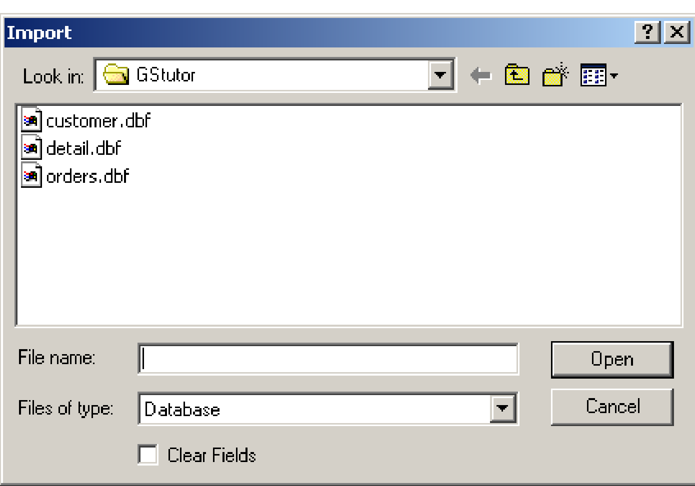
Importing the Database File

The DB Server Editor provides an easy way for you to get a quick start on defining a data server. If you have an existing database file to work with, you can simply import it into the editor.

To import a .DBF file:

1. Click on the Find button to the right of the File Name edit control.

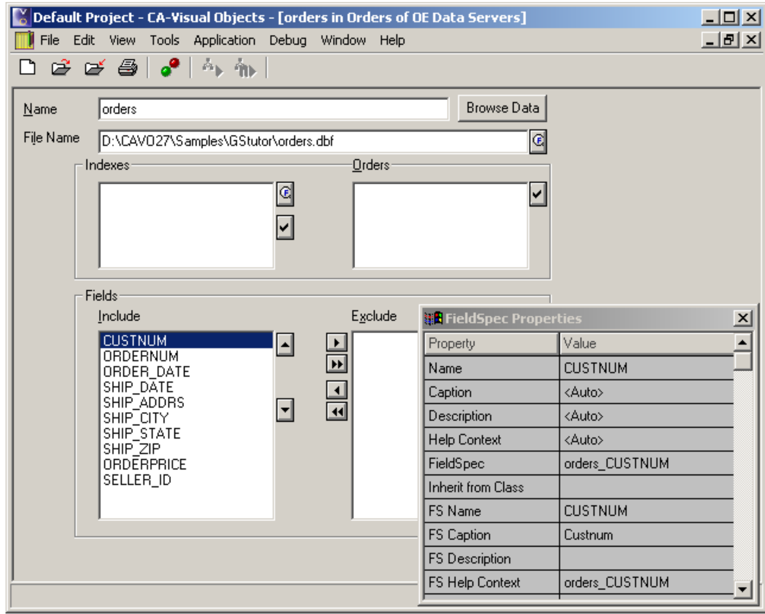
   A standard Import dialog box for .DBF files appears:



2. Choose ORDERS.DBF from your Visual Objects \SAMPLES\GSTUTOR directory, and click Open.

   Information about the selected file is immediately imported into the editor.

For example, a default name is defined for the data server entity, based on the name of the file you selected; the path and name of the selected .DBF file are displayed in the File Name control; and all the fields defined in the structure of the .DBF file are listed in the Include list box:



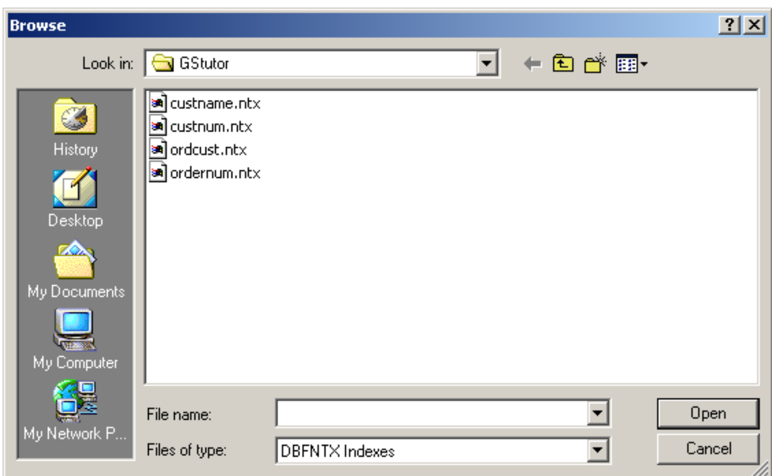
Note: By default, when you first import a .DBF file into this editor, *all* fields in the .DBF file are placed in the Include list box. The fields are presented in the Include list box in the same order as they appear in the database file header.

Importing the Index Files

You can just as easily import the two index files associated with ORDERS.DBF. To do so:

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

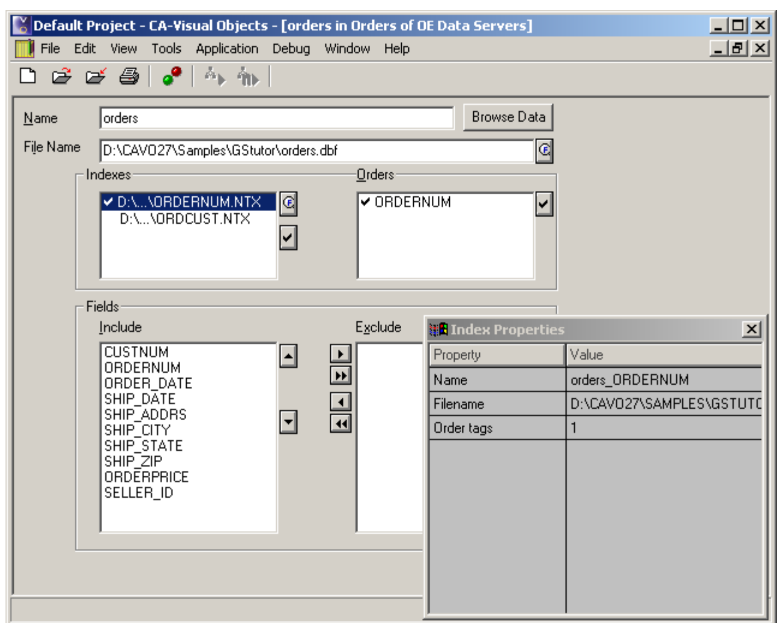
   This presents you with a standard Browse dialog box for index files—in this instance .NTX files:



2. Choose ORDCUST.NTX and ORDERNUM.NTX, located in the \SAMPLES\GSTUTOR directory, by pressing Ctrl and clicking on each file.

3. Choose the Open button.

The selected index files are imported and now appear in the Indexes list box:



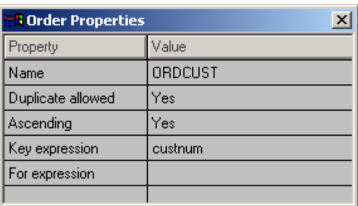
Controlling Order

Because more than one index was selected, the order that they are imported in is not guaranteed. The first one that the editor imported will be marked as the controlling order.

Make sure that ORDCUST.NTX is checked in the Indexes list box, indicating that this file contains the controlling order. If not, highlight it in the Indexes list box and click on the check mark button to the right.

Notice that since an index file currently has focus, the Properties window displays index-related properties. The Order Tags property indicates the number of orders within an index, which is of interest if you are using multi-order index files. In this case, we are using .NTX files, which support only one order per file, so ORDCUST.NTX shows only one tag, ORDCUST, which appears in the Orders list box.

Now click on ORDCUST in the Orders list box. The Properties window changes, displaying order-related properties:

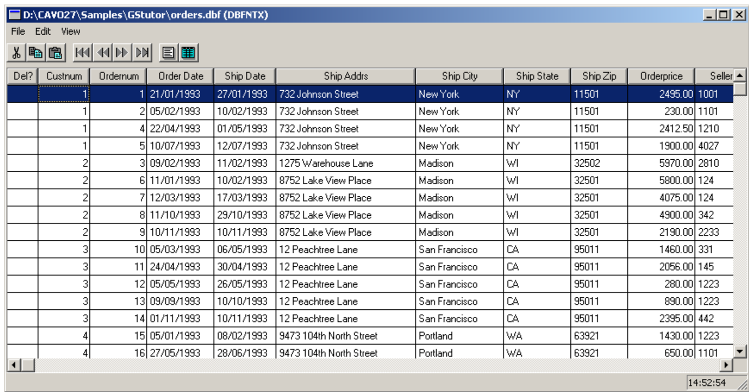


As you can see, the key expression for ORDCUST is CustNum. We'll use this fact in the next lesson, when we set up a master-detail relationship between the Customers and Orders data servers using a data window.

Browsing Data

Before customizing the data server lets see what our data looks like. Choose the large Browse Data button at the end of the Name field.

This will display the current .DBF file that has been imported to the DB Server Editor:



Del?	Custnum	Ordernum	Order Date	Ship Date	Ship Addrs	Ship City	Ship State	Ship Zip	Orderprice	Seller
	1	1	21/01/1993	27/01/1993	732 Johnson Street	New York	NY	11501	2495.00	1001
	1	2	05/02/1993	10/02/1993	732 Johnson Street	New York	NY	11501	230.00	1101
	1	4	22/04/1993	01/05/1993	732 Johnson Street	New York	NY	11501	2412.50	1210
	1	5	10/07/1993	12/07/1993	732 Johnson Street	New York	NY	11501	1900.00	4027
	2	3	09/02/1993	11/02/1993	1275 Warehouse Lane	Madison	WI	32502	5970.00	2810
	2	6	11/01/1993	10/02/1993	8752 Lake View Place	Madison	WI	32501	5800.00	124
	2	7	12/03/1993	17/03/1993	8752 Lake View Place	Madison	WI	32501	4075.00	124
	2	8	11/10/1993	29/10/1993	8752 Lake View Place	Madison	WI	32501	4900.00	342
	2	9	10/11/1993	10/11/1993	8752 Lake View Place	Madison	WI	32501	2190.00	2233
	3	10	05/03/1993	06/05/1993	12 Peachtree Lane	San Francisco	CA	95011	1460.00	331
	3	11	24/04/1993	30/04/1993	12 Peachtree Lane	San Francisco	CA	95011	2056.00	145
	3	12	05/05/1993	26/05/1993	12 Peachtree Lane	San Francisco	CA	95011	280.00	1223
	3	13	09/09/1993	10/10/1993	12 Peachtree Lane	San Francisco	CA	95011	890.00	1223
	3	14	01/11/1993	10/11/1993	12 Peachtree Lane	San Francisco	CA	95011	2395.00	442
	4	15	05/01/1993	08/02/1993	9473 104th North Street	Portland	WA	63921	1430.00	1223
	4	16	27/05/1993	28/06/1993	9473 104th North Street	Portland	WA	63921	650.00	1101

You can use the Navigation buttons to scroll through the data and switch between browse and form view using the toolbar buttons. After browsing the data close the window. You are now returned to the DB Server Editor.

Sharing Field Specifications

The next step, after importing the database and index files, is to customize some of the default field properties that were automatically created by Visual Objects for the fields in the new data server. For the most part, you'll be changing the default field captions so that they are more descriptive, and you'll be adding status bar descriptions.

Before we go on to these tasks, however, we'll "pull in" some of the field specifications that are already defined in the Customer data server. We can do this because both the CUSTOMER and ORDERS files contain fields pertaining to customer number, state, and zip code.

The Customer data server was *predefined*: all required properties had been filled in for you, and the default field specifications for the CustNum, State, and Zip fields had already been customized.

Therefore, since the two .DBF files share common fields, we can reuse the field specifications already defined for the Customer data server in our new data server, instead of defining them over again.

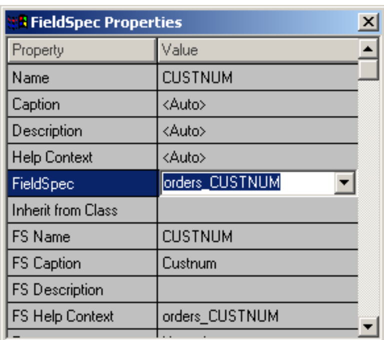
Let's start with the CustNum field:

1. Click on the CustNum field in the Include list box and then click on the Properties dialog to change focus to it..

   This field is identical to the CustNum field in the Customer data server, so we can take advantage of work we've already done by using its CustNum field specification.

2. Scroll through the Properties window to see what the DB Server Editor defines for you automatically.

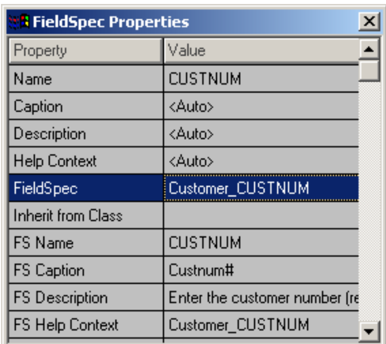
3. When you're through browsing, click on the FieldSpec property in the Properties window.



4. Click on the Down arrow, scroll through the list until you see Customer_CustNum, and then click on it.

   In this step, you are taking properties from the field specification defined for the CustNum field in the *Customer* data server and simply reusing them in the *Orders* data server.

   For example, the FS Description property in this server is updated with the text defined in the other data server (this property was blank before):



   Or, if you scroll down the list, you can see that the Required and Validation properties defined for the CustNum field in the Customer data server have been imported. (These properties were also blank before.)

Note: For new and updated field properties, see the FieldSpec Properties Window topic in the online help.

Pulling in the existing field specification does all this work for you—that's all you need to do for this field.

5. Repeat the steps listed above to associate the Ship_State field with the Customer_State field specification and the Ship_Zip field with the Customer_Zip field specification.
6. Click on the Save button in the DB Server Editor toolbar to save your work so far.

You may be wondering why there appear to be two Caption and Description properties. This is because both of these properties have a hierarchical nature in Visual Objects.

Briefly, the Caption and Description properties (at the top of the Properties window) are associated with a field via its *hyperlabel* (described in The Source Code section later in this lesson). However, each field also has FS Caption and FS Description properties that are associated with its *field specification*.

In both cases, the two caption-description pairs serve a common purpose: the *caption* is used in the data window you'll create later to label the field, and the *description* is displayed in the owner window's status bar when the field has focus.

By default, the system will first use the FS Caption and FS Description properties defined for a field; the Caption and Description properties are provided in case you want to override the corresponding "FS" properties.

Note: This hierarchy is described in greater detail in the "Using the Window Editor" chapter of the *IDE User Guide*.

Customizing Field Properties

The next task is to define some properties for the other remaining fields.

OrderNum

Let's start with OrderNum:

1. Click on the OrderNum field in the Include list box.
2. In the Properties window, click on the FS Caption property. Clear the current contents and type in the text **Order #**, and press Enter.
3. Click on the FS Description property, type **Enter the order number (required)**, and press Enter.
4. Scroll down to the Required property, click on it, click on the Down arrow, and choose Yes from the list box.

   Changing this property to Yes will require that the user type a value for the field in the resulting application.
5. Click on Required Diagnostic just below, type **You must enter an order number**, and press Enter.

   If the user attempts to skip the OrderNum field, this message will be displayed.
6. Scroll down to Validation, click on it, type **{ |OrderNum| OrderNum > 0}**, and press Enter.

   This will require that the user type a positive order number. The validation rule specified here is in the form of a code block. Refer to the "Code Blocks" chapter in the *Programmer's Guide* for more information.
7. Click on Validation Diagnostic just below, type **The order number must be positive**, and press Enter.

   If the user attempts to enter a negative number or zero into the OrderNum field, this message will be displayed.
8. Click on the Save toolbar button.

OrderPrice

For the OrderPrice field, follow these steps.

1. Click on the OrderPrice field in the Include list box.
2. Click on the FS Caption property, edit the current contents to read **Order Price**, and press Enter.
3. Click on FS Description, type **Enter the order price**, and press Enter.
4. Scroll down to the Picture property, click on it, type **$$$$$$$.99**, and press Enter.

   This will cause the value to display with leading dollar signs.

5. Optionally, you can continue to customize the remaining fields in the data server by adding status bar descriptions and/or customizing the default captions to make them more readable (for example, for the Ship_Addrs field, you might change “Ship Addrs” to “Shipping Address”).

   If desired, however, you can skip this task.

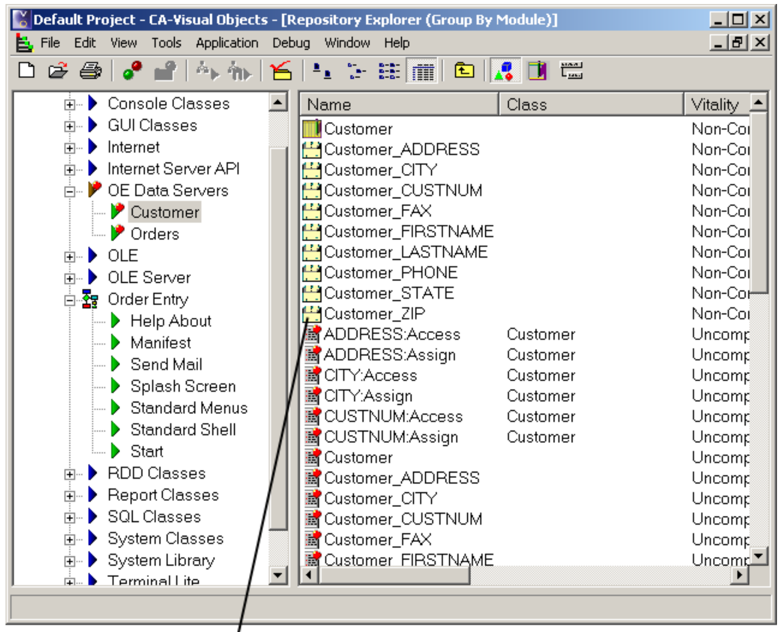
6. Click on the Save toolbar button.

The FieldSpec Editor

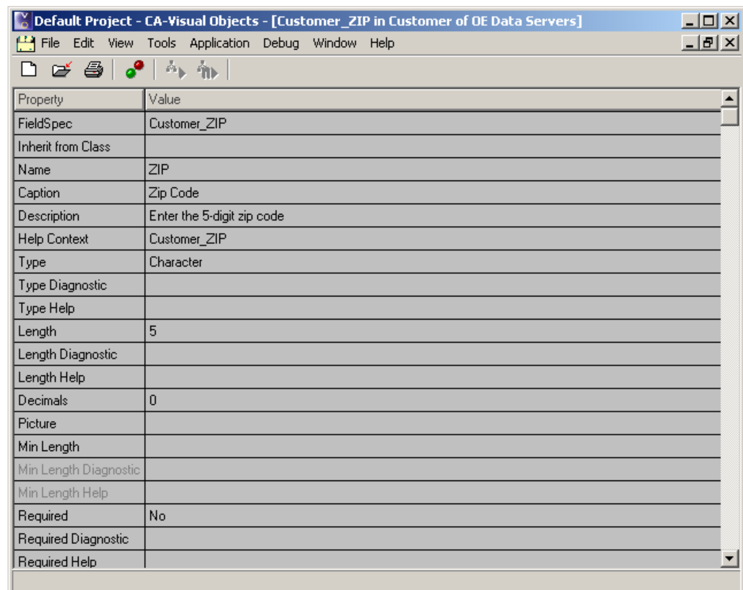
Now, you will get a chance to experience the power of using field specifications. What you'll do is use the FieldSpec Editor to make a small change to one of the field specifications shared by the two data servers. You can then watch how the one change is automatically propagated to the two data servers using that field specification.

Before starting on this task, close the DB Server Editor and save your changes if prompted.

Now, click on the Customer module (displaying its entity listing in the Repository Explorer's list view pane). Next, click on the Type column heading to display the entities by type, (you may need to scroll the list view to see the Type column). Now you can easily find the Customer_Zip field specification entity and double-click on it:



This invokes the FieldSpec Editor, as shown below:



As you can see, the workspace of the FieldSpec Editor is remarkably similar to the Properties window of the DB Server Editor when a field is selected. The only difference is that there is only one entry for the field-specific properties—that is, Name, Caption, Description, and HelpContext are not present.

The FieldSpec Editor is provided primarily to let you create new field specifications, independent of a particular data server, which can then later be associated with fields in a data server.

You can also use the FieldSpec Editor to modify existing field specifications. If there are any existing entities in the system already using field specifications that you modify here (for example, a data server or a data window), the changes you make in the FieldSpec Editor are automatically propagated to those entities.

Let's explore this last in more detail. In our two data servers (Customers and Orders), the Customer_Zip field specification is used to format their individual zip code fields. The Customer data server you imported already contained the Customer_Zip field specification, and you then associated that same field specification with the Ship_Zip field in the Orders data server.

Modifying the Field Specification

Let's make a small change to this shared field specification in the FieldSpec Editor:

1. Scroll to the Picture property and click on it.
2. Type **99999** and press Enter.
3. Close the FieldSpec Editor and save your changes.

Viewing the Automatic Change Propagation

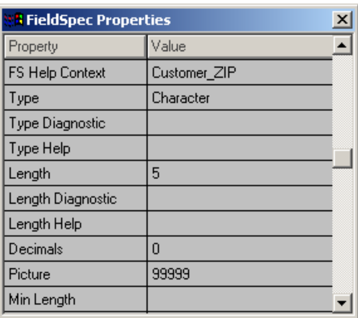
Now let's see how this one change has been propagated automatically in the two data servers that use this field specification:

1. In the Customer module's list view, double-click on the Customer server entity.

   The DB Server Editor is loaded, and the Customer data server is opened.

2. In the Include list box, click on the Zip field.
3. Click in the Properties window and scroll through it, until the Picture property comes into view.

   You will see the picture clause you just entered (which was not there before).



Note: For new and updated field properties, see the FieldSpec Properties Window topic in the online help.

4. Close the DB Server Editor.
5. Now click on the Orders module and then double-click on the Orders server entity to open the Orders data server. Take a look at its Ship_Zip field's Picture property, and you will see that it has changed too.

Since both data servers pick up the Picture property from the Customer_Zip field specification entity you edited, both reflect the change. If any other data servers used this field specification, they would also reflect the change. In addition, if there were any data windows using either of these data servers, the windows would also pick up the change. Visual Objects takes care of all of this automatically.

That's all there is to creating a data server. If you have not already done so, close all copies of the DB Server Editor currently in use, and return to the Repository Explorer entity listing for the Orders module. From there, we'll take a closer look at the source code generated by the DB Server Editor for the Orders data server you just created.

The Source Code

As you may have been observing, each time you choose the Save toolbar button or the File Save menu command while in an editor, simply designing pieces of an application in an editor and then saving your work automatically causes Visual Objects to generate code. This code can be modified in the Source Code Editor if desired, but typically you will use the editors to make changes and then regenerate code when you save the changes.

Nevertheless, even though you probably won't be directly modifying the generated code, it may help to take a closer look at exactly what was just generated for you, so that you get a better understanding of how all of the generated source fits together.

Tip: You can follow this discussion by scrolling through the entity listing to view the various entities or, if you like, you can double-click on the entities to view them in the Source Code Editor.

Server and Field Specification Entities

To start with, when you design a data server in the DB Server Editor, Visual Objects creates a single *server* entity. There is also a *field specification* entity for each field in the data server. The purpose of this design is to let you easily edit the entire data server with the DB Server Editor or just an individual field specification in the FieldSpec Editor.

Class Entities

There are also *class* entities for the data server and each of its field specifications. If you double-click on the Orders class entity, for example, you will see that it inherits from the DBServer class, which is defined in the RDD Classes library. Notice, too, that the associated database file for this data server—cDBFPath—is defined as an instance variable of the Orders class as part of this inheritance.

Each field specification class entity inherits from the FieldSpec class (this class is defined in the System Classes library).

Access/Assign Entities

Next, there are *access/assign method* entities for each field, which provide an object-oriented interface to the database fields.

Init() Method Entities

Finally, there are *Init() method* entities for the data server and each of its fields.

For the data server, the Init() method does the following:

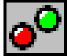
- Sets up a file specification for the database file.
- The FileSpec class helps you manage files, keeping track of information such as the name and location of the file (called the *file specification*). (See the "File Handling" chapter in the *Programmer's Guide* for more specific information.)
- For each field, defines a hyperlabel, instantiates the appropriate field specification class, and sets up a data field using the DataField class
- The HyperLabel class helps you keep track of certain information associated with an object (called the object's *hyperlabel*), such as a name, caption, description, and help context ID. (See the "Hyperlabels" chapter in the *Programmer's Guide* for more specific information).
- Sets up a file specification for each index file and opens the files in the appropriate order.

For the field specifications, the Init() methods define hyperlabels for each field specification and assign values to the various properties.

Now that you have imported the Customer data server and set up the Orders data server, the OE Data Servers library is complete. There are only two steps left to complete this lesson: building the OE Data Servers library and associating it with the Order Entry application. Before continuing, close the Source Code Editor, if you are using it to view the generated code, and choose No, if prompted to save changes.

Building the OE Data Servers Library

Anytime you import a new library or make changes to an existing one, it is necessary to build the library to make sure that there are no errors in the source code and to make it available in compiled form to any applications that use it.



You've already gone through the process of building the Standard Application in the previous lesson—building a library is no different. Simply click on the Build button when the library has focus, and you're done.

Adding the Library to Order Entry's Search Path

After it is built, you need to add this library to the search path of the Order Entry application. In Visual Objects, an application's properties include its *name*, *type* (that is, executable, library, or DLL), and its associated *libraries*, among other things.

Recall that in the first lesson of this tutorial, you initially specified these items for the application when you created it. However, you can change an application's properties at any time using the Application Properties menu command.

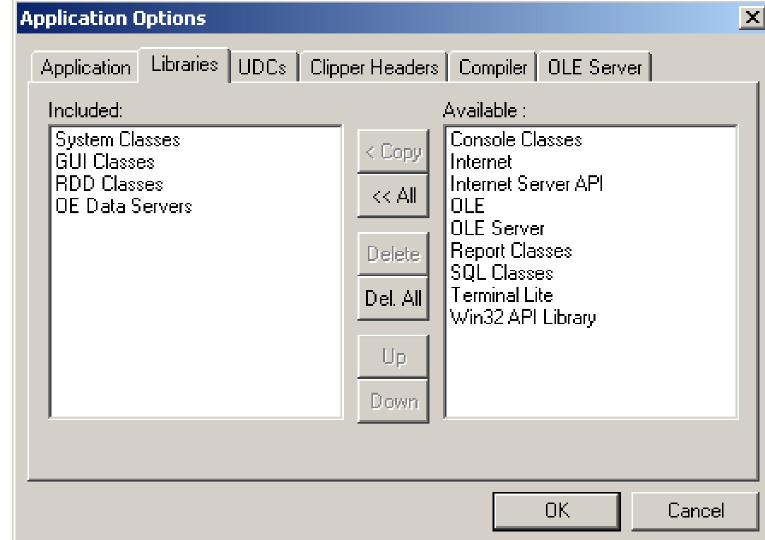
Let's use this command now to add the OE Data Servers library to the Order Entry application's path:

1. If you haven't already done so, build the library by clicking the Build button.
2. Select the Order Entry application in the Repository Explorer's tree view.
3. Choose the Application Properties menu command.

   The Application Options dialog box appears.
4. Click on the Libraries tab.

   This will display all the libraries available.
5. Add the OE Data Servers library to your application by double-clicking on it in the Available list box.

OE Data Servers is then added to the Included list box.



6. Choose OK.

7. Rebuild the Order Entry application by clicking the Build toolbar button.

Summary

That concludes the lesson in setting up data servers. You've taken a big step in customizing the Order Entry application, but you won't be able to see the results of what you've done until you set up a data window that uses these data servers. Therefore, let's move on to the next lesson and do just that.

Lesson 3: Creating a Data Window

In this lesson, you will use the Window Editor to create a data window. As explained earlier, a *data window* is a type of *window* with which you can associate one or more data servers. Data *windows* are preconfigured so that they know how to display and operate on the information extracted from their underlying data servers.

Master-Detail

For the Order Entry application, we'll create a *data window* that "links" the Customer and Orders data servers in a *master-detail* fashion. Recall that CustNum is a field that is common to both the Customer and Orders data servers. In addition, the Orders data server uses the CustNum field as its controlling order key.

Thus, the two data servers can be linked together, based on the contents of this common field, in master-detail fashion. When data servers are linked in this fashion, the detail data server is synchronized with the master data server: the detail data server is automatically repositioned whenever the master data server moves to a new record. Thus, the data windows containing the two data servers will display only orders whose customer number matches that of the current customer record.

Note: The link between the two data servers is established using the DataWindow:SetSelectiveRelation() method.

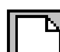
Auto Layout

To create the data window, you'll use the Window Editor's Auto Layout feature. Using this feature will give you a productive start on your data window design and will also demonstrate how a great deal of the information you defined for the two data servers is automatically picked up by the data window.

Starting the Window Editor

To create a data window for the Order Entry application:

1. Click on the Order Entry application in the Repository Explorer.

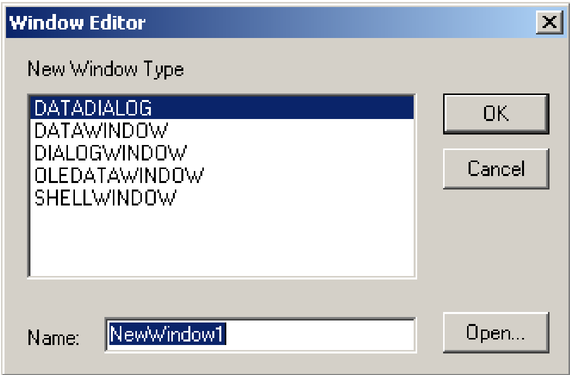


2. Click on the New Module button, and in the Create Module dialog box, type **App Windows**, and choose OK.

   Note: If you don't remember where the New Module button is, simply move the mouse over the toolbar to display the tooltips for the toolbar items.

3. Click on the **App Windows** module in the Repository Explorer. Then click on the New Entity toolbar button, and from the local pop-up menu, choose Window Editor.

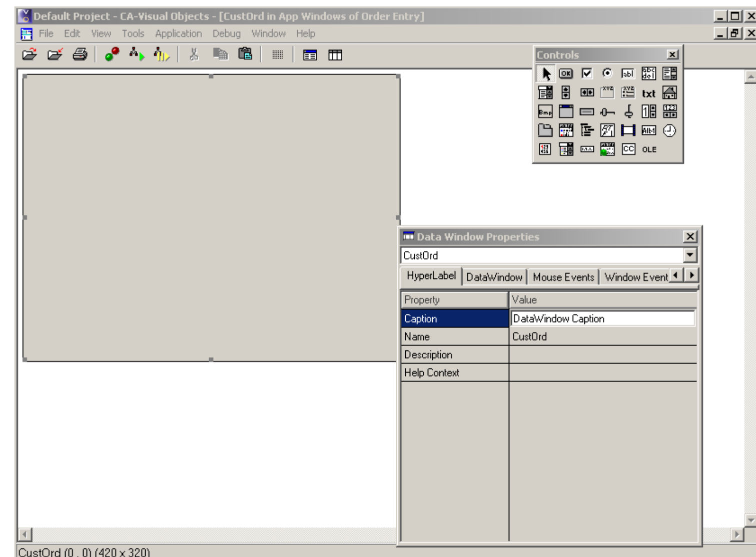
   The following dialog box is displayed; it allows you to choose what type of window to create, as well as enter a name for it:



4. Since you are creating a data window, click on the DATAWINDOW selection, then type **CustOrd** in the Name edit control.

5. Choose OK to launch the Window Editor.

Your desktop should now look something like the following:



Tip: If desired, reposition the Properties window and/or tool palette.

When you first access the Window Editor for a new window, you see an empty form template on which you can place controls, a floating tool palette from which to select the type of controls to place, and a Properties window to define properties for the data form and the various controls.

Note: The Window Editor creates a binary form entity, so when referring to the data window in the Window Editor it will be referred to as a *data form*.

Window Properties

The Properties window probably looks familiar to you, because it is very similar to the one you used when working with the DB Server Editor. Like the DB Server Editor's Properties window, this one also changes depending on what currently has focus in the editor (for example, a form or a control).

Initially, the data form's blank template is selected, and, therefore, the Properties window displays properties for the data form. Let's take a look at some of the properties available to a data form.

Caption

The first thing you see in the Properties window drop-down list is the caption of the data form. This caption will appear in the title bar of the data window at runtime.

Name

This is the name of the data form, CustOrd. This is the name that you entered earlier when you launched the Window Editor.

Click on the DataWindow tab of the Properties window to display the next set of properties. Use the scroll bar to view the properties that are not in view.

View As

The View As property controls how the window displays data when it is initially opened. You can choose either form view or browse view, which you are familiar with from working with the data windows in the Standard Application back in the first lesson of this tutorial.

Data Server

Data Server is also a property of a data form. Like View As, this property will be filled in by the Auto Layout feature later in the lesson.

Menu

Now let's take a look at the Menu property. If you remember from working with the Standard Application, each child data window had its own menu (StandardShellMenu) that replaced the shell window's menu (EmptyShellMenu) when the child window was open and had focus.

Using the Menu property, we will give our new data window this same behavior:

1. With the DataWindow tab control selected, click on the Menu property.
2. Click on the Down arrow button. If StandardShellMenu appears in the list, choose it from the resulting list box, and press Enter. If StandardShellMenu does not appears in the list, type **StandardShellMenu** into the edit area, and press Enter.

The StandardShellMenu is now attached to the CustOrd data window. Note that StandardShellMenu's toolbar is automatically associated, too.

Caption

You also need to define a caption for the CustOrd window. A window's caption appears in its title bar.

1. Select the Hyperlabel tab control and click on the Caption property.
2. Type **Customer Orders** and press Enter.

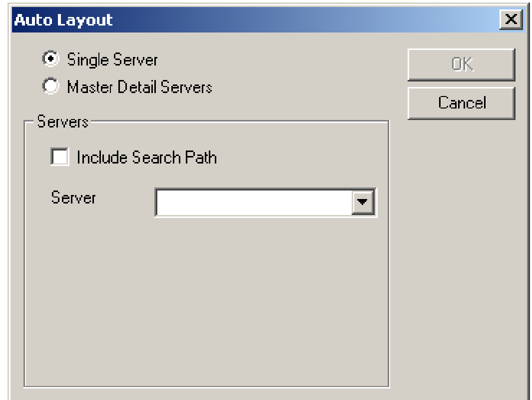
As you can see, there are several more window properties that we have not discussed, but for this data window, the default values for these properties will suffice.

Using Auto Layout

Now, we want to move on and experiment with the Auto Layout feature to get the form started:

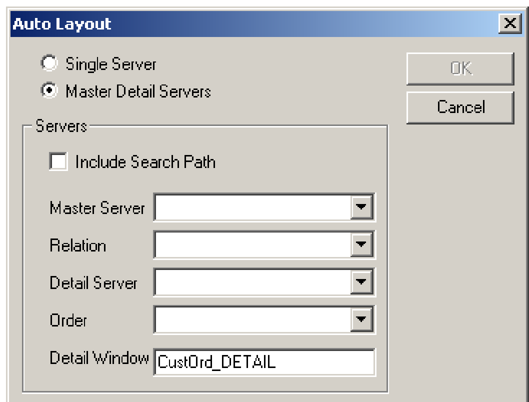
1. Click on the Auto Layout toolbar button:

The Auto Layout dialog box appears:



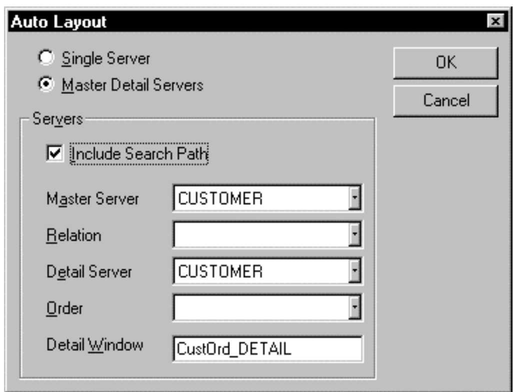
2. Click on the Master Detail Servers radio button.

This changes the dialog box slightly, as shown below:



3. Since both of our data servers are defined in the OE Data Servers library, rather than in the Order Entry application itself, click on the Include Search Path check box.

   This will cause the Customer data server to appear in both the Master Server and Detail Server combo boxes:

4. In the Relation combo box, click on the Down arrow button—a list of the fields in the Customer data server appears—and choose **#CUSTNUM** (the common field).

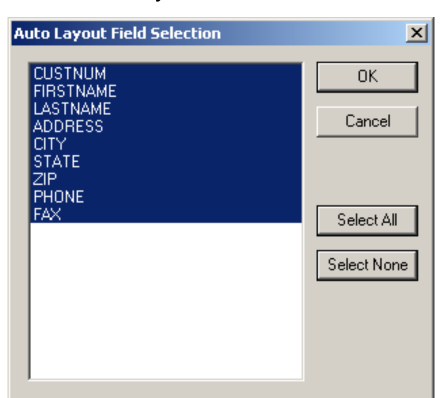
   Thus far, the master data server and the common field are already defined. Now set the detail data server.

5. In the Detail Server combo box, click on the Down arrow and choose **Orders**.

6. In the Order combo box, click on the Down arrow and choose **ORDCUST**, which is the controlling order we specified earlier when we imported the index files for the Orders data server.

7. Choose OK.

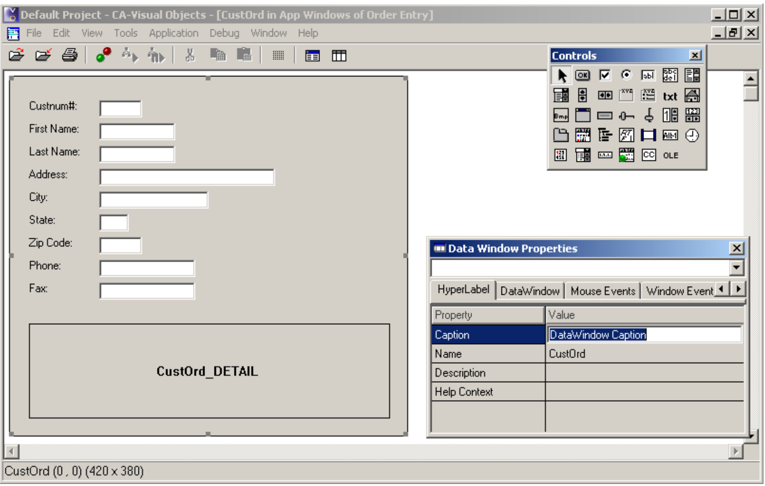
The Auto Layout Field Selection dialog box appears:



This dialog box allows you to select the fields that you want to be displayed on the window.

8. For our application we will select all of the available fields, since this is the default, click OK.

A default data form layout is created, using information you already defined for the Customer and Orders data servers:



When you use Auto Layout to create a master-detail window, you actually get *two* data windows. We'll refer to them, while in the Window Editor, as the main data form and the sub-data form.

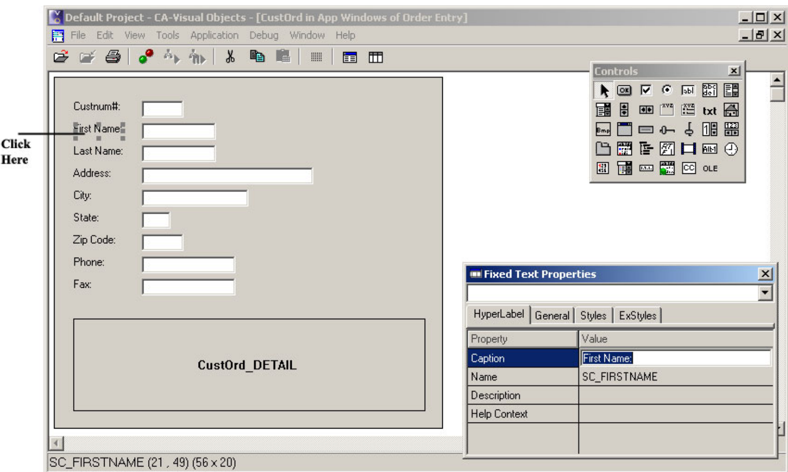
A Closer Look at the Main Data Form

You will notice that when you create a data window from a customized data server, many of the properties defined in the associated data servers are used by the Window Editor to create the controls in the data form.

Captions Reused

For example, the Window Editor uses the caption defined for each field in the master data server to create each field's fixed text control (either the Caption or FS Caption property, as defined in the data server).

To see what we mean, click on the First Name label and watch what happens to the Properties window:



This control functions as a decorative label for the edit control to its right.

Now, click on the single-line edit control to the right of the First Name label:

This edit control represents the field itself—you can see that the properties defined for its associated data server field while in the DB Server Editor are reused here.

Notice the Caption property: this caption is not used by the single-line edit control because the main data window is currently in form view (if you recall from Lesson 1, a data window can be displayed in either form view or browse view). If you were to switch to browse view (by clicking the Browse/Form View toolbar button), you would see that the caption is used for the heading in the corresponding column control. (You'll witness this later on in this lesson.)

Descriptions Reused

Another example of how the Window Editor reuses data server properties is its use of the descriptions it picks up from the underlying data servers. Since a single-line edit control is currently selected, you can see in the Properties window that the Description property of that control contains the description of its corresponding data server field. Later on, when one of these controls has focus in the resulting application, this information will be displayed on the status bar of the shell window.

Field Specifications Reused

Noticed that each single-line edit control picked up the field specification from its corresponding data server field.

When you run the new application, the picture clauses and validation rules that you defined for the field specifications will be enforced by the program when the data window is displayed. For example, the numbers in the Order Price field will be formatted accordingly, and the user will be warned if they attempt to enter an order number less than zero.

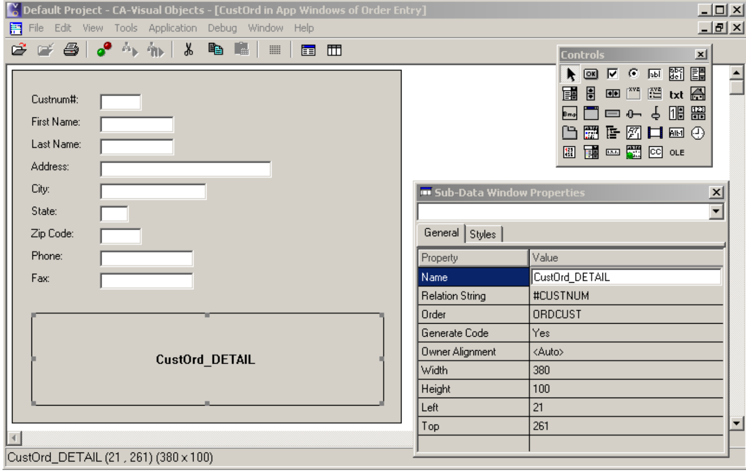
A Closer Look at the Sub-Data Form

Before we explore the sub-data window in more detail, let's first resize the main data form, so that the sub-data form is completely visible (unless it is already in full view). To do this, first click on the Window Editor's bottom edge and drag it downward to expose the bottom edge of the main data form. Then click the sizing handle in the lower right-hand corner of this form, and drag it down until the sub-data form is fully displayed—all standard Windows techniques, by the way.

Tip: If necessary, move the Properties window and the tool palette out of the way by clicking on their title bars and dragging them to new positions.

The main CustOrd data form now displays the entire CustOrd_DETAIL sub-data form:

If you now click on the CustOrd_Detail sub-data form, its properties are displayed in the Properties window. There are nine listed on the General tab page, including Name, RelationString, Order, and Generate Code.



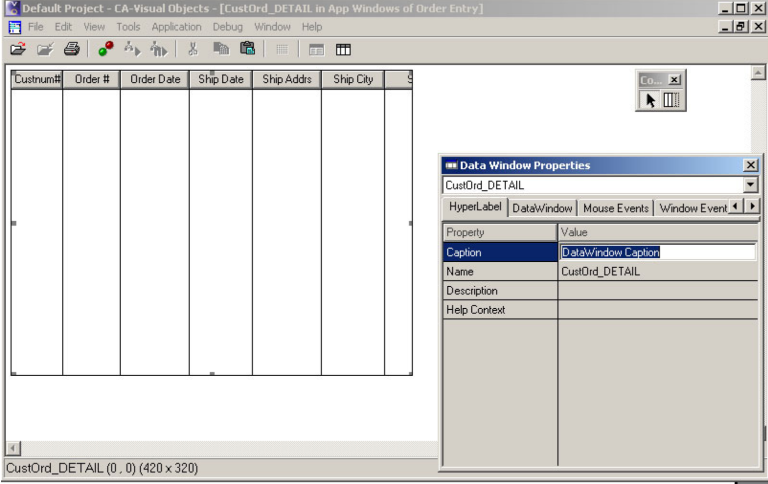
The *name* is inserted by the Auto Layout feature and indicates that the sub-data form represents the detail data server in the CustOrd data window. The *relation string*, also inserted by the Auto Layout feature, represents the common field used to relate the sub-data window's underlying data server (Orders) with the main data window's associated data server (Customer). The Order property reflects the specified *controlling order*.

Additionally, the Styles tab allows you to select the Tab Stop option from the Sub-Form Styles dialog box in order to enable the tab key for the sub-data form. Lastly, the Generate Code option indicates whether or not source code will be generated for the sub-data form control.

Opening the Sub-Data Form

There are, in fact, additional properties that are available to the sub-data form and to view them, you should double-click on the sub-data form now. Doing so actually launches a new copy of the Window Editor, unique to the sub-data form.

As you can see, you get both a new Properties window and a new tool palette, which allows you to add columns:



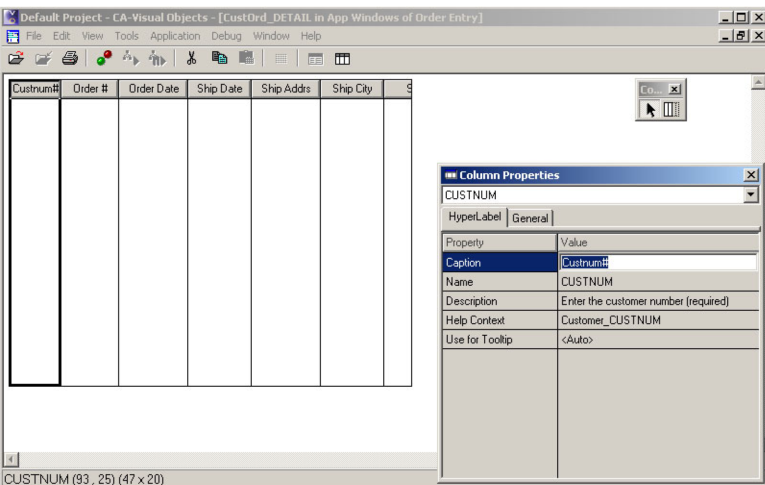
There are many more properties available now—you may notice that this sub-data form is associated with the Orders data server and its default view is browse.

Browse View

In Lesson 1 you learned that when a data window is displayed in browse view, it displays each field in the underlying data server as a column. This is in contrast to form view (the view used for the main data form) in which each field in the underlying data server gets a fixed text control and an edit control.

Note that the column headings in this browse view are extracted from the captions defined for each field in the underlying data server (reusing your work in the DB Server Editor).

This is because, like the single-line edit controls in the main data window, these columns represent the fields themselves. If you click on a column heading, you can see that the properties defined for its associated data server field while in the DB Server Editor are reused here:



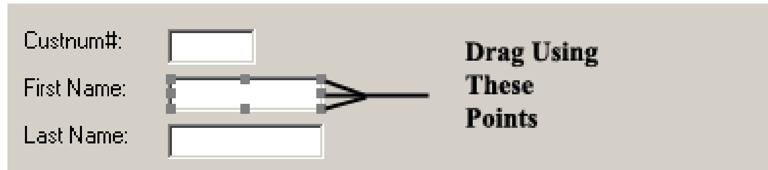
Finally, note that viewing the sub-data form in this workspace gives you an idea of how that rectangular control in the main data window will look at runtime. (The rectangular control is used for the size and placement; this view is used to illustrate the column heading text, spacing, etc.) Close the sub-data form by double-clicking on its system menu and choose No if you are prompted to save changes.

Customizing Windows

The master-detail window that you just created simply by using the Auto Layout feature is fully operational and ready to use. You may, however, choose to modify it by resizing it or moving or resizing any of its controls, or by changing its properties or the properties of any control associated with it.

Sizing Windows and Controls

For example, you've already resized the main data form by clicking and dragging on one of its sizing handles so that the sub-data form would be fully visible. Similarly, you can resize its controls using the same technique:



Note: With Some controls, such as radio buttons, although the overall control can be resized, parts have a fixed size that you can not change.

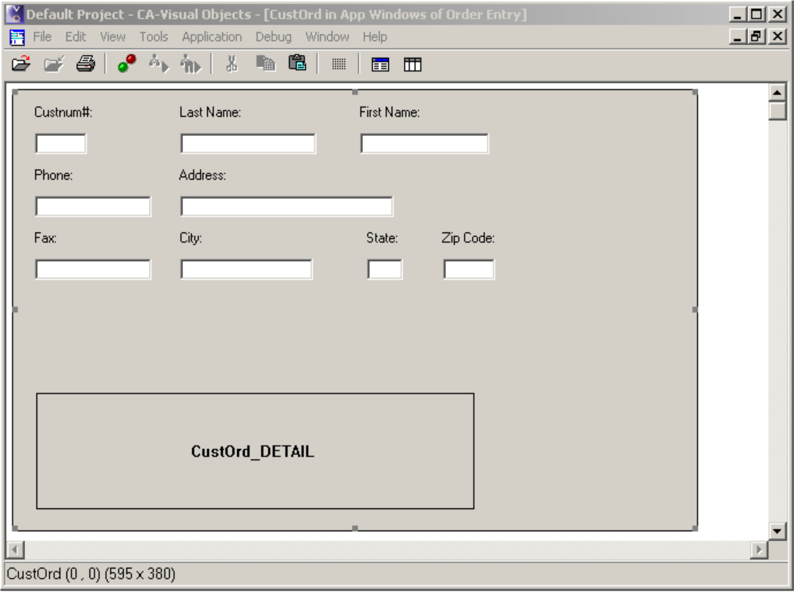
Moving Controls

Perhaps you would like the controls to be arranged across the top of the main data window, with Last Name placed before First Name, since most customer records are maintained in alphabetical order by last name. Also, you might like Phone and Fax placed under Custnum#.

First, if necessary, resize the form to allow the controls to fit in the form horizontally. To make arranging the controls easier:

1. Place the mouse pointer anywhere on a control.
2. Press the left mouse button and hold it down.
3. Drag the mouse to move the control to the desired location, and release the mouse button.
4. Repeat steps 1–3 for all fixed text and edit controls.

The CustOrd main data form should now look something like this:



Notice that there is now more room for the sub-data form, so you can even resize that, if you like.

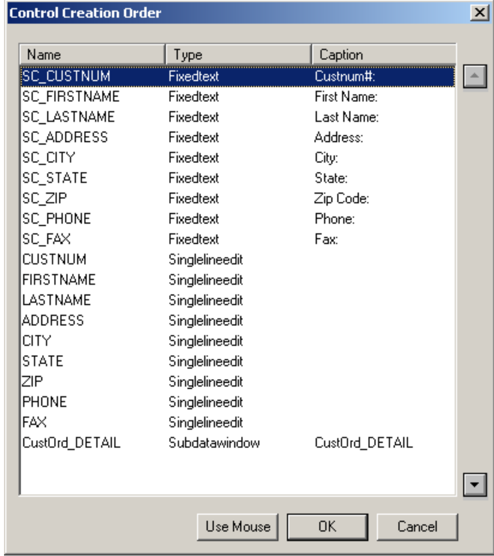
Positioning Controls

There are a number of useful commands on the Edit Arrange menu that will allow you to arrange these controls so that they have the same alignment, size, or spacing. These include such commands as Align Left, Align Right, Center Vertically, Center Horizontally, Even Vertical Spacing, and so on. See the Modifying a Window in the "Using the Window Editor" chapter of the *IDE User Guide* for more information about moving, resizing, and positioning controls.

Viewing Tab Order

When you add controls to a form, the system automatically creates a default tab order for cursor movement. The initial order is based on the vertical and horizontal position of the controls. The control in the upper left-hand corner comes first, and subsequent controls are ordered based on a left-to-right, top-to-bottom progression.

To view the tab order for a data form, choose the Edit Control Order menu command. For example, below is the default tab order of the CustOrd main data form *before* we made any changes:



The Control Creation Order dialog box displays the names of all the controls, as well as their captions and control types, in *tab stop* order as they appear in the source code. It also allows you to change the cursor tabbing order for a window's controls by reordering the controls in the source code.

The Use Mouse button allows you to modify the tab order interactively, rather than using the more traditional Up and Down arrow buttons. For more detailed information, see the online help.

When you customize a data form by rearranging the controls, the system automatically updates the default tab order until you save the editor. Once you have saved the design, you are responsible for maintaining the order.

The Control Creation Order dialog box allows you to change the default tab order that is set automatically by the system when you place your controls.

Note: This command affects only the order in which the cursor moves from control to control within a window; it does *not* alter the controls' actual positions on the window.

For more detailed information, see Changing Tab Order by Reordering Controls in the "Using the Window Editor" chapter of the *IDE User Guide*. Also, refer to the online help.

Moving On

You could make other modifications, such as changing the color or font used for any control in either the main data form or its nested sub-data form using the appropriate Properties window. However, our main data form is fine. Close the Window Editor—saving the changes, of course—and look at the resultant source code.

The Source Code

Similar to the DB Server Editor, saving your changes in the Window Editor generates source code. Although you should never directly modify the generated code, it may help to take a closer look at exactly what was just generated for you. This will give you a better understanding of how all of the generated source code fits together.

Window Entities

First, there are *form* entities for both the CustOrd and the CustOrd_Detail windows. The purpose of these entities is so that you can easily edit the data windows with the Window Editor.

Class Entities

There are also class entities for both windows that inherit from the DataWindow class defined in the GUI Classes library. These windows can, therefore, be edited and used individually.

Resource and Constant Entities

There are resource entities for both windows, along with several constant entities to number the individual edit controls. In order to see these constants use the View Options menu command and choose the Entity View tab. Under Hidden Items, select the Show All Items option and choose OK.

Access / Assign and Init() Method Entities

Finally, there are access/assign methods for the field edit controls and Init() methods for both of the DataWindow subclasses.

These Init() methods are complex, defining all edit controls in the window's form view and instantiating a HyperLabel object for each one. The field edit controls also have an appropriate FieldSpec object attached.

Then, the data window is attached to the appropriate data server with the Use() method. If you switch from form view to browse view, the browse view for the window is defined with the DataBrowser and DataColumn classes (also defined in the GUI Classes library).

Finally, the window establishes its default view with the ViewAs() method and, in the case of the CustOrd window, attaches the detail window and establishes the relationship between the windows using the SetSelectiveRelation() method mentioned earlier.

Summary

That concludes the lesson in setting up the data window. You will now move on to modify the EmptyShellMenu to add a menu command that will open this data window.

Lesson 4: Modifying the Menu

In this lesson, you will use the Menu Editor to customize the EmptyShellMenu entity of the Order Entry application. To this existing menu structure, you will add a new menu command, which will be used in the main window (StandardShellWindow) of the resulting application to open the Customer Orders data window you just created.

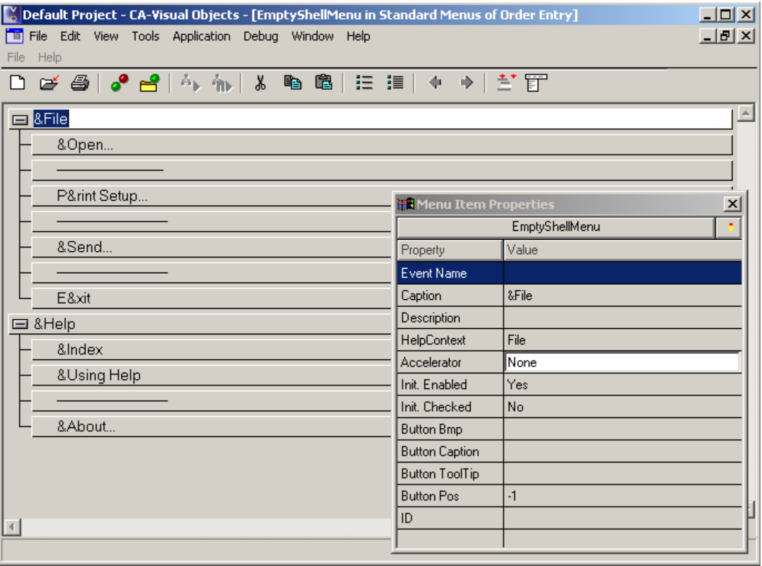
Note: As you probably recall, EmptyShellMenu and StandardShellWindow were generated as part of the Standard Application back in the first lesson.

Starting the Menu Editor

To modify this menu:

1. Click on the Standard Menus module in the Repository Explorer in order to view its entities.
2. Double-click on the EmptyShellMenu binary menu entity.

The Menu Editor is launched—your desktop should now look as follows:



The menu structure for the empty shell menu appears as a tree structure in the Menu Editor workspace, and the preview menu bar shows the two existing menus, File and Help, across the top of the window under the normal menu.

Adding the Customer Orders Menu Command

We're going to add a menu item, the Customer Orders command, to the File menu just below the existing Open command:

1. Click on the &Open menu item:
2. Press Enter.

   This opens up a blank line directly below the &Open menu item in which you can define the new menu command.
3. In the blank line, type **C&ustomer Orders...**.

   This is the name of the menu command as it will appear in the File menu. Note, however, that the ampersand (&) character will not appear in the menu; instead, it causes the letter "u" to be underlined in the menu, indicating it as the key for selecting the command.

   Note: The "..." is a Windows convention used to indicate that a menu command displays a dialog box or window.
4. Next, click on Event Name in the Properties window, type **CustOrd**, and press Enter.

   A menu command's Event Name property is used to define the action that should occur when the user chooses the menu command in the resulting application. We want the program to display the Customer Orders data window—therefore, all we need to do is supply the name of the data window (note that CustOrd is the data window's *name*, while Customer Orders is its title, or *caption*).

   That's all the program needs to know to open the window when this menu command is chosen.
5. Finally, click on the Description property, type **Open the customer orders window**, and press Enter.

This description will appear in the status bar when the menu command is highlighted.

More on Event Handling

Our work in the Menu Editor is just about complete. Before we move on, however, let's explore in more detail exactly how the menu command will display the Customer Orders data window. If you'll remember, we discussed event handling in the first lesson of this tutorial, including the hierarchy within the automatic event-handling system. This is an important feature of Visual Objects so it bears repeating.

By default, when the system encounters an event name, it first looks for a method name that matches the event name. If none is found, it then looks for a window name that matches. Finally, if no matching method or window name is found, it looks for a report. Thus, by assigning CustOrd to the event name, choosing the File Customer Orders menu command will automatically display the Customer Orders data window.

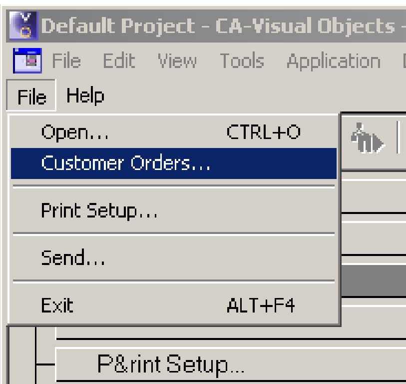
This feature makes it easy to connect events to menu commands and window controls (using the Menu and Window Editors, respectively). You simply enter the name of the method, window, or report that you want to activate as the Event property (as you did in the steps above), and the rest is taken care of automatically.

Previewing Your Work

At any time, you can select the entries in the Menu Editor's preview menu bar (just as you would a real menu) to preview what your menus look like.

Note, however, that the preview menu bar is only partially operational. That is, it allows submenus to be pulled down, but nothing actually happens if you select a menu command. Its principal intent is to provide visual feedback while you are designing a menu structure.

To take a look at your new menu command, click on the File menu in the preview menu bar—you'll see the new Customer Orders menu command:



Summary

Your menu is done. Double-click on the system menu to close the Menu Editor and choose Yes when prompted to save your work. The majority of the source code for this menu was reviewed in the first lesson, but if you look at it now, you will see several new constant entities for the new menu command you just added.

As you have seen, the Menu Editor is very straightforward and easy to use. In this lesson, you've learned a key principle behind the automatic event handling inherent in all Visual Objects applications—event handling by name. In the next lesson, you will use this feature once again to link two new methods to commands that you will add to the StandardShellMenu.

Lesson 5: Adding the Ordering Methods

If you remember, when you viewed the Customer data server earlier, it has two associated orders, CUSTNUM.NTX and CUSTNAME.NTX. As a final enhancement to the Order Entry application, we'll create methods to switch the controlling order for the Customer data server between these two.

To accomplish this, we will first add two commands to StandardShellMenu's View menu, "By Name" and "By Number," and link them to events named "ByName" and "ByNum," respectively. (These events represent the two methods, ByName() and ByNum(), to be associated with the two menu commands—more about these later.)

Note that because the By Number menu command represents CUSTNUM.NTX, which is the default controlling order, it will be checked when the menu is initially displayed, indicating that the records in the data window are initially sorted by number.

We're adding the two new menu commands to StandardShellMenu, so they're going to be displayed as soon as the user opens one or more .DBF files in the application. However, these menu commands are specifically designed to work with the Customer Orders window and should only be available when that window is open.

Therefore, we'll disable the By Name and By Number commands when we create them, so that when StandardShellMenu is first displayed, the commands appear grayed, or dimmed. We'll then add code to the CustOrd window's Init() method to enable the commands when the Customer Orders data window is opened.

Note: An *enabled* menu command can be selected by the user. A *disabled* menu command appears dimmed and cannot be selected; it remains unavailable until it is enabled by the application.

We will also import two methods, named ByName() and ByNum(), which contain the necessary code to change the controlling order for the data displayed in the Customer Orders window. ByName() changes the controlling order to CUSTNAME.NTX, and ByNum() changes it to CUSTNUM.NTX. Note that these are the *event-handling methods* associated with our two new menu commands; these methods also contain code to check (and uncheck) the appropriate menu command so that the View menu always reflects the current order in the window.

Modifying the Menu

Let's start, then, by customizing StandardShellMenu to add the two new menu commands. If you do not have the Standard Menus module selected, click on this module in the Repository Explorer. Then double-click on the StandardShellMenu menu entity to load it into the Menu Editor.

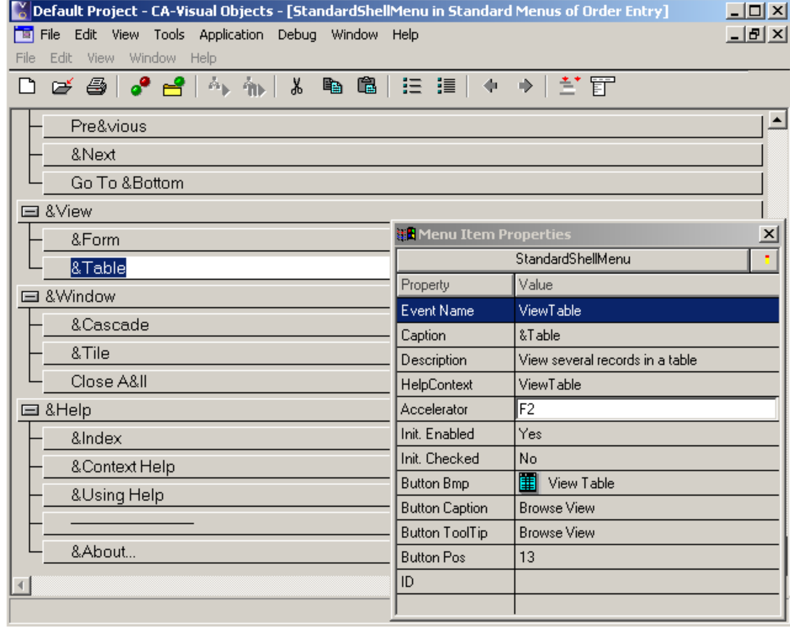
Adding Commands

Since the methods that we are going to add change the manner in which the data is displayed, it makes sense to put the commands on the View menu.

Note that the two new menu commands are distinct from the other commands on the View menu. Therefore, we're going to add a separator to the menu before we actually add the commands. (In Windows applications, separators are used to logically group items within a menu.)

Follow these steps to insert a separator and the two new menu commands at the bottom of the View menu:

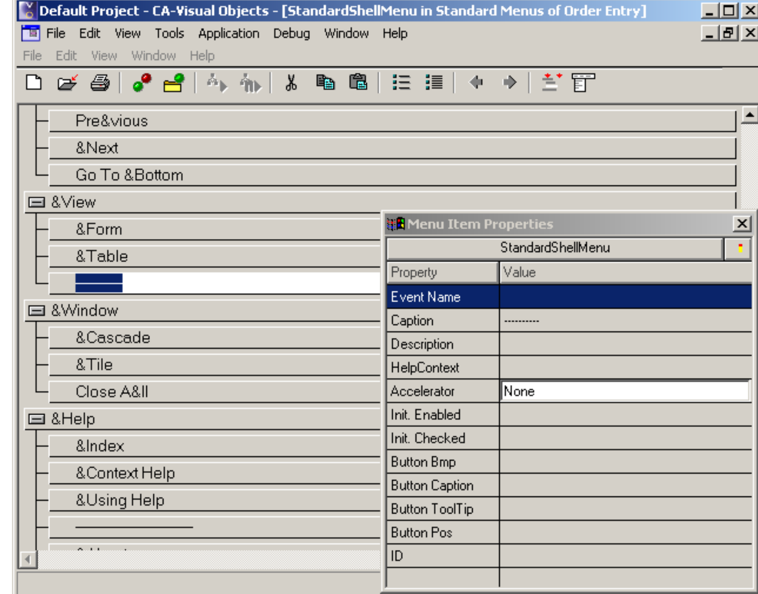
1. Use the scroll bar in the Menu Editor to scroll down to the View menu, and then click on the &Table menu item:



2. From the topmost Edit menu, choose the Add Item command, and then choose Add Separator from the resulting pop-up menu.

   Note: Remember, the menus in the lower menu bar are just prototypes of the menus you are creating in this editor.

This immediately adds a separator to the menu structure:



3. Press Enter to create a new, blank menu item, and type **By &Name** in the new line.

4. Press Enter again, and type **By Nu&mber** in the next new line.

The new menu commands are now part of the menu structure; next, you need to define properties for them.

Defining Menu Properties

In order to operate properly, the program needs to know what event names the new menu commands are associated with, as well as what their initial status should be. We'll also define status bar descriptions for them.

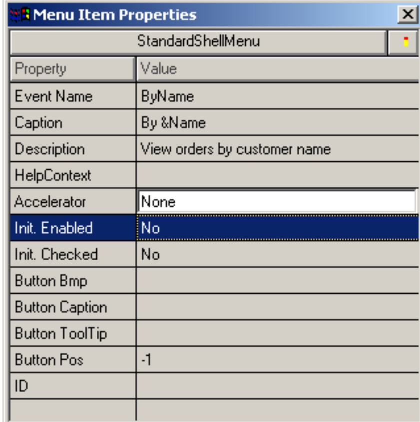
By Name

Let's start with the By Name menu command:

1. Click on the By &Name menu item in the menu structure.
2. In the Properties window, click on Event Name, type **ByName**, and press Enter.

   This is the name of the method that this menu command should be associated with; it will switch the controlling order to CUSTNAME.NTX. (You'll import this method later.)
3. Click on Description, type **View orders by customer name**, and press Enter.
4. Click on the Init. Enabled property, click on the Down arrow, choose No from the resulting list box, and press Enter.

   This changes the value of this property to "No," thereby causing the By Name menu command to be disabled when the menu is first displayed. (Later, we'll add code to the CustOrd:Init() method to enable this menu command when the CustOrd data window is open.)



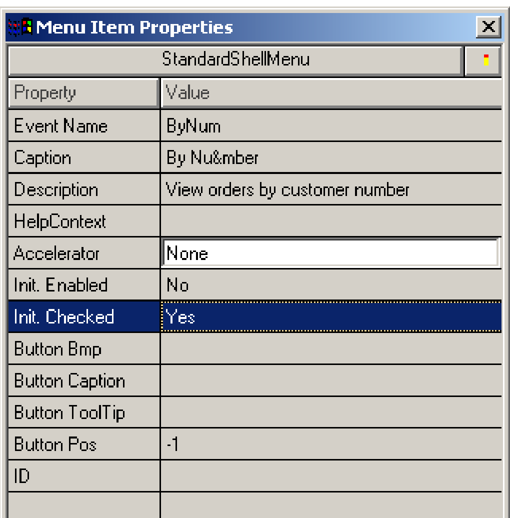
By Number

Now, you will follow similar steps for the By Number menu command:

1. Click on the By Nu&mber menu item in the menu structure.
2. In the Properties window, click on Event Name, type **ByNum**, and press Enter.

   This is the name of the method that this menu command should be associated with; it will switch the controlling order to CUSTNUM.NTX. (You'll import this method later.)
3. Click on Description, type **View orders by customer number**, and press Enter.
4. Click on the Init. Enabled property, click on the Down arrow, choose No from the resulting list box, and press Enter.

   Like the By Name menu command, By Number applies only to the CustOrd data window, so it will be disabled when the menu is initially displayed. (Later, we'll add code to the CustOrd:Init() method to enable this menu command when the CustOrd data window is open.)
5. Click on the Init. Checked property, click on the Down arrow, choose Yes from the resulting list box, and press Enter.



The default controlling order for the CustOrd data window is CUSTNUM.NTX. Therefore, the By Number menu command should be checked when the CustOrd data window is initially displayed. (The ByName() and ByNum() methods that we'll import later will contain code to appropriately toggle the check mark between the two menu commands.)

That's all you need to do to the menu (if you like, you can use the preview menu bar to see the new commands). When you are finished, close the Menu Editor and save your work.

Note: You will not be able to save your menu entity if there are any *empty* menu items. Use the Edit Delete Item menu command to delete a blank menu item, if necessary.

Creating the Methods

The next step is to add the functionality for the By Name and By Number menu commands. Since both the data server on which the methods will operate and the menu to which the methods are attached are owned by the CustOrd class, we'll add the methods to the App Windows module in the Order Entry application.

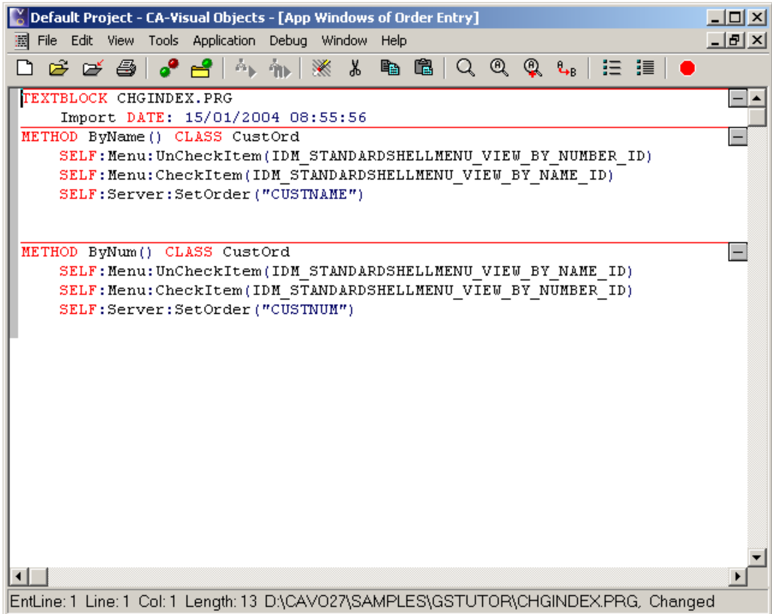
To do this:

1. Select the App Windows module in the Repository Explorer.
2. Click on the New Entity toolbar button and choose Source Code Editor from the local pop-up menu.

   The Source Code Editor window displays and is empty.
3. Choose the File Import command.

   This presents you with a standard Import dialog box for .PRG files.
4. Choose the file named CHGINDEX.PRG from your Visual Objects \SAMPLES\GSTUTOR directory.

   The code from the .PRG file is imported into the Source Code Editor:



There are three entities in this source code: a TEXTBLOCK statement and two methods belonging to CLASS CustOrd. (The CustOrd class was created for you by Visual Objects when you saved the CustOrd window in the Window Editor.)

The TEXTBLOCK Entity

The TEXTBLOCK entity is automatically inserted by Visual Objects every time you import a source file to denote the name, date, and time of the imported file.

The Methods

The two methods, ByName() and ByNum(), are just a few lines long:

```
METHOD ByName() CLASS CustOrd
   SELF:Menu:UnCheckItem(IDM_STANDARDSHELLMENU_VIEW_BY_NUMBER_ID)
   SELF:Menu:CheckItem(IDM_STANDARDSHELLMENU_VIEW_BY_NAME_ID)
   SELF:Server:SetOrder("CUSTNAME")

METHOD ByNum() CLASS CustOrd
   SELF:Menu:UnCheckItem(IDM_STANDARDSHELLMENU_VIEW_BY_NAME_ID)
   SELF:Menu:CheckItem(IDM_STANDARDSHELLMENU_VIEW_BY_NUMBER_ID)
   SELF:Server:SetOrder("CUSTNUM")
```

The source code for these two methods is nearly identical. Basically, the first one changes the controlling order to CUSTNAME.NTX, and the second changes it to CUSTNUM.NTX.

Let's take a line-by-line look at this code, using the ByName() method as an example, to see exactly what these methods do.

```
METHOD ByName() CLASS CustOrd
```

This statement declares the ByName() method as belonging to the CustOrd class.

```
SELF:Menu:UnCheckItem(IDM_STANDARDSHELLMENU_VIEW_BY_NUMBER_ID)
```

This method call unchecks the View By Number command in the StandardShellMenu. If you remember from reviewing the source code for both menus in this application, each option has several define constants associated with it. One of these is a unique number that identifies the menu item. In this case, IDM_STANDARDSHELLMENU_VIEW_BY_NUMBER_ID is the unique identifier for the View By Number command.

The UnCheckItem() method is defined in the Menu class (all menus created in the Menu Editor automatically inherit from the Menu class). The method takes the unique identifier as an argument, so it knows which menu item you are referring to. Finally, "SELF:Menu" returns the menu owned by this data window, directing the UnCheckItem() call to the proper object.

Note: SELF:Menu:***MethodCall*** is the standard construction that you will use to direct a method invocation to the owned menu of any window. You will see it again later in this lesson when modifying the CustOrd:Init() method.

```
SELF:Menu:CheckItem(IDM_STANDARDSHELLMENU_VIEW_BY_NAME_ID)
```

This method call checks the By Name command on the View menu, using code almost identical to that described above.

Note: Since the ByName() method is called when the View By Name menu command is chosen, it will remove the check mark displayed next to the By Number command and place a new check mark next to the By Name command. This way, the menu will always reflect the correct controlling order.

```
SELF:Server:SetOrder("CUSTNAME")
```

This method call changes the controlling order to CUSTNAME.NTX. This is accomplished using the SetOrder() method (which is defined in the DB Server class, found in the RDD classes library from which our data servers inherit).

Similar to the manner in which the data window's menu was identified, "SELF:Server" identifies its data server, properly directing the SetOrder() call.

Note: SELF:Server:***MethodCall*** is the standard construction that you will use to direct a method invocation to the owned data server of any window.

Now, close the Source Code Editor, saving the imported code, and we'll move on to the final step in this lesson. What we have done so far is sufficient to add the menu commands to the StandardShellMenu and define their functionality, but since both menu commands are initially disabled, we could not use them if we stopped here.

Enabling the Menu Commands

The reason behind initially disabling these menu commands was that the StandardShellMenu is shared by many different data windows, and none of them, besides CustOrd, have the associated index files necessary to properly utilize these commands. Our strategy was to disable them by default and enable them only when they apply (that is, when CustOrd is open).

The CustOrd class has an associated Init() method that is called whenever the class is instantiated, and the class is instantiated only when the CustOrd window is opened. This is the correct time for the menu items to be enabled but remember that we said, "You must never edit generated code".

The reason you must never edit generated code is simple. If you make manual changes to the generated code and save it, then go into the editor, make some changes and save them, the changes that you made manually will be lost and you will need to make them again. However, you will often need the window to react differently as it is being instantiated, just as we need it to now.

To allow for the tailoring of windows, menus, servers and in fact almost all objects, Visual Objects automatically provides calls to two methods from the Init methods of objects, the PreInit() and PostInit() methods. These two methods are called by the generated code, one before the main code is run and one after it has finished and easily provide us with an opportunity to tailor our window without the dangers described above

To customize the CustOrd instantiation, method:

1. Click on the AppWindows module in the Repository Explorer to display its entities.
2. Click on the New Entity toolbar button and choose Source Code Editor from the local pop-up menu.

   The Source Code Editor window displays and is empty.

3. Type the following line of code:

```
METHOD PostInit( ) CLASS CustOrd
```

4. Press Enter to move to the next line and, type the following line of code:

```
SELF:Menu:EnableItem(IDM_STANDARDSHELLMENU_VIEW_BY_NAME_ID)
```

5. Press Enter again, and type the next line of code:

```
SELF:Menu:EnableItem(IDM_STANDARDSHELLMENU_VIEW_BY_NUMBER_ID)
```

The first line of code declares that this is a method named PostInit and that it belongs to the CustOrd class. The other two lines of code enable the View By Name and By Number menu commands for the CustOrd window, similar to the way in which the ByName() and ByNum() methods checked and unchecked the menu commands.

Because the enabling of these menu options takes place in a method of the CustOrd window, they will no normally be available to any other window that may use this same menu class.

To save this new code, double-click on the system menu and choose Yes when prompted.

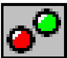
Summary

With just a few lines of source code and a slight change to the StandardShellMenu, you've added quite a bit of functionality to the Order Entry application. You've learned some valuable tips, both on managing menus—by exploring some of the methods defined for the Menu class (such as EnableItem() and CheckItem())—and accessing methods defined for the data server (such as SetOrder()).

Both of these techniques were accomplished using the ownership relationships between the window and its menu and data server, and the knowledge of these relationships by the data window object through its Menu and Server properties. So, you've also learned a lot about where to put the source code to manage events (such as menu commands) and are probably feeling pretty comfortable about how all of the components in the Order Entry application fit together.

Next, you will build Order Entry and see all of your customized features in a working application.

Lesson 6: Running the Order Entry Application



At this point, you've completed the tutorial, but you still haven't seen your Order Entry application in action. So, build the new Order Entry application now (just click on the Build button, as always).

Generating an Executable File

So that you can complete the development cycle, this time, instead of running the new application from within the IDE (like you did for the Standard Application in the first lesson), you will generate an executable file and run it from Windows.

To do this, click the Make EXE toolbar button.

A dialog window will be displayed to that you can see something is happening and, when this closes, Visual Objects will have generated the Order Entry.EXE file.

When the Standard MDI Sample was imported to start with, the path to the .EXE was pointing at %ExecutableDir% which is the system variable for the C:\CAVO27\Bin directory. Therefore, the OrderEntry.Exe will have been created in this directory. If you have changed that path then it will be wherever the Application Properties path is pointing.

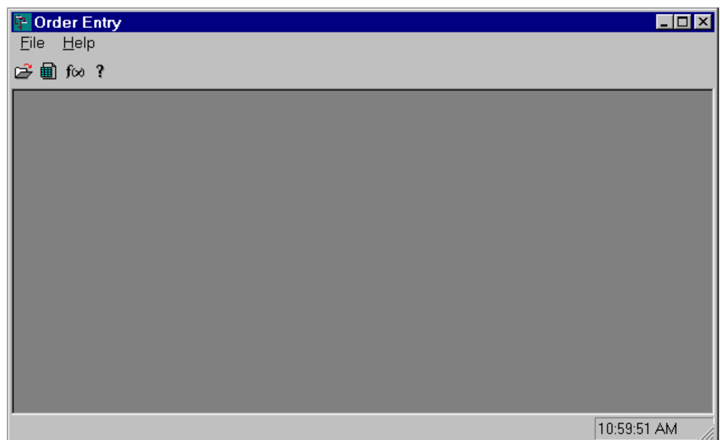
Note: Remember that the executable file name and folder were specified in the Properties dialog box when you originally created this application.

Running the Application

Using the Windows Explorer, go to the C:\CAVO27\Bin directory and locate the OrderEntry.Exe file.

You can now run the Order Entry application by simply double clicking on the OrderEntry.exe.

After a short display of a splash screen, it is loaded as follows:

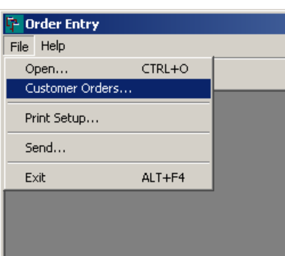


Looking at the New Features

Let's take a look at some of the new features you added to the application.

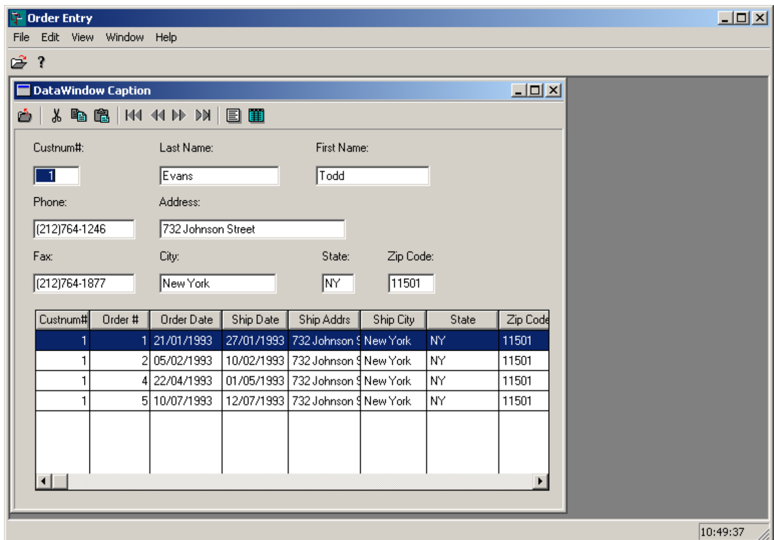
Menu Changes

For example, if you open the File menu, you will see the Customer Orders command that you added to the EmptyShellMenu:



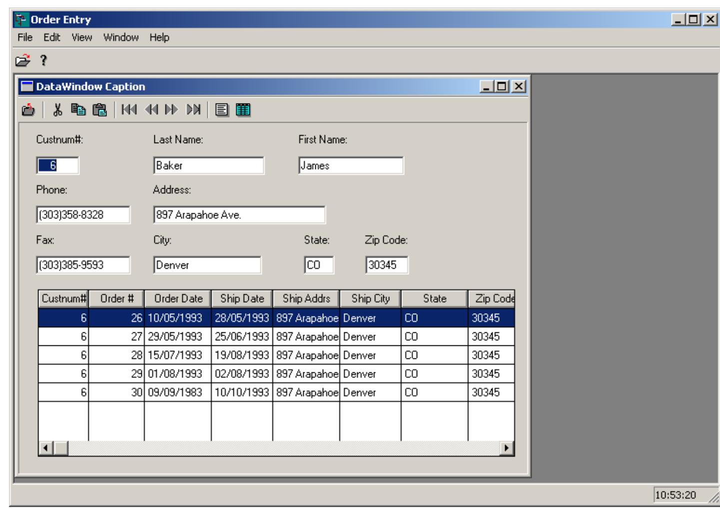
Master-Detail Window

Choose the Customer Orders command, and you will see the Customer and Orders database servers displayed in the master-detail data window that you designed. If you then resize the shell window and the Customer Orders window so that the sub-data window is completely visible, your screen should look similar to the following:



Sub-Data Window

Scroll through the customer records using the navigation buttons on the toolbar, and watch as the orders sub-data window is updated to reflect the current customer. You will also notice that the customer records are in order according to the customer number, which is due to the fact that the CUSTNUM.NTX index is defining the controlling order:



Picture

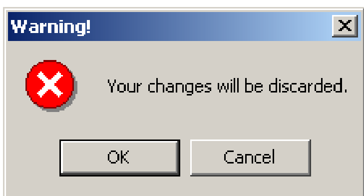
If you scroll the orders window until the Order Price column comes into view, you will see the picture formatting that you defined in the Orders data server:

Zip Code	Order Price	Seller Id
30345	$$$1990.00	114
30345	$$$$125.00	4256
30345	$$$2330.00	13346
30345	$$$$695.00	2452
30345	$$$1951.00	44223

Validation

Scroll the sub-data window back to the left until the Order # field comes into view. You may remember that when you created the Orders data server, you specified a validation rule and diagnostic message for this field to prevent negative numbers. Now, double-click on the field, type **-1**, and press Tab to move to the next field.

Although the validation rule cannot prevent you from moving to a new field, if you now click on the Go Next navigation button, you will see the validation diagnostic message “The order number must be positive” and this warning box:



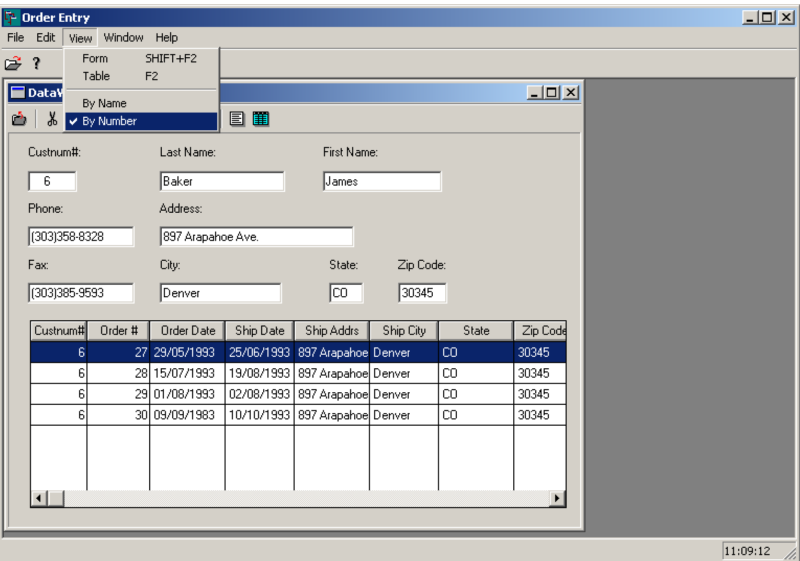
Also, if you attempt to close the file at this point, the invalid data will not be propagated down to the actual database.

Description

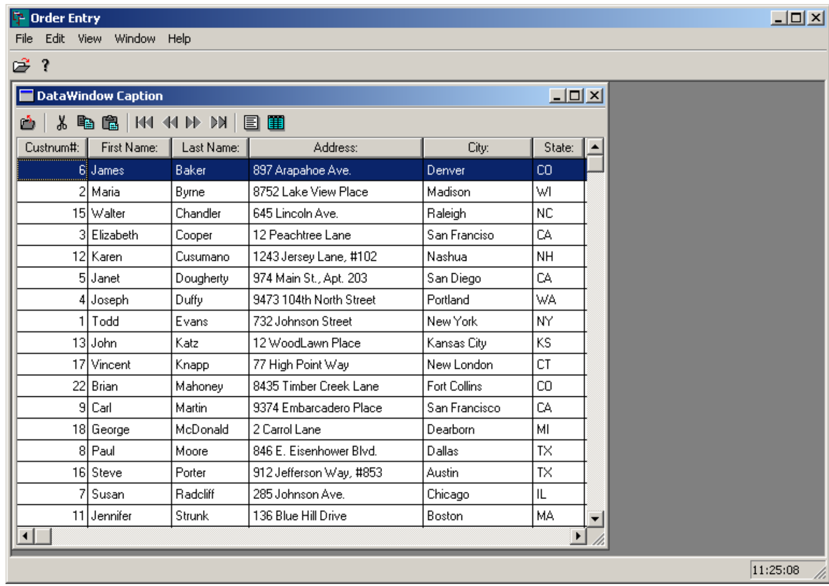
Move the cursor out of the sub-data window to one of the other field controls (such as Custnum #). Look at the status bar, and you will see the description for the field with focus. Click on another field, and watch as the status bar is updated with a new description.

Controlling Order

Now open the View menu, and you will see the By Name and By Number commands, with By Number checked:



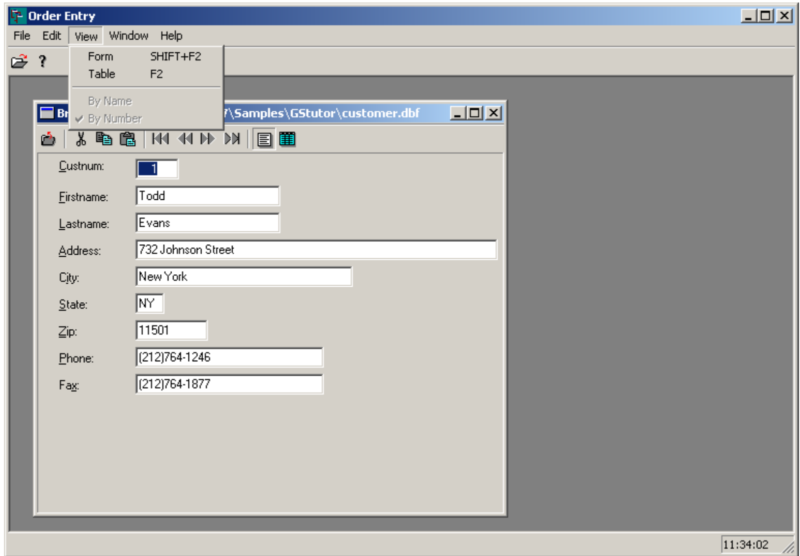
Choose By Name, and the controlling order for the Customer data server will change to CUSTNAME.NTX. Choose the View Table command, and you can easily see that the customers are now in order by last name instead of customer number:



Take a look at the View menu again—you will see that the By Name menu command is checked. (If desired, choose By Number to return to the previous controlling order.)

The last part that we need to look at is not in this window so, close the CustOrd window by double-clicking on the system menu. You may be warned that there is invalid data, which will not be saved; if so, choose Yes.

Now, using the ordinary File Open command, open any Dbf file. When the file is open in a data window, open the View menu again. Notice that the By Name and By Number commands are grayed out, because we made sure that they would only be available for the CustOrd window:



When you have filished examining the application you can close it in the usual way.

What's Next

That concludes this tutorial of Visual Objects. You have learned some important concepts regarding the structure of an MDI application and have seen how the Standard Application fits into that structure and provides a starting point for your own customized application.

In just a few lessons, you have experimented with most of the visual tools available in the IDE and have seen how they interact with one another through the source code that they generate. You have also written and linked in some source code of your own.

Now that you have learned how to create, compile, and execute a Visual Objects application, you may want to go on to the *IDE User Guide*, which provides more detailed information about the IDE, especially the Repository Explorer and visual editors, editing and debugging applications, and creating .EXEs. Or, to learn more about programming in Visual Objects, read through the *Programmer's Guide*.